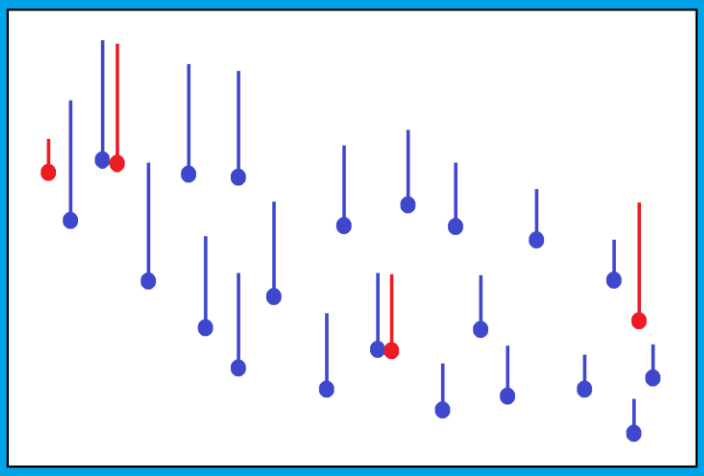
Mirosław Kordos

Instance Selection in Machine Learning

New Directions in Similarity-based, Evolutionary and Embedded Methods



Akademia
Techniczno-Humanistyczna
w Bielsku-Białej
2019

WYDZIAŁ BUDOWY MASZYN I INFORMATYKI

Mirosław Kordos

Instance Selection in Machine Learning

New Directions in Similarity-based, Evolutionary and Embedded Methods

BIELSKO-BIAŁA 2019

Redaktor Naczelny: prof. dr hab. n.t. Iwona Adamiec-Wójcik
Redaktor Działu: prof. dr hab. inż. Jacek Stadnicki
Recenzenci: prof. dr hab. inż. Tadeusz Wieczorek
dr hab. inż. Rafał Scherer, prof. PCz.
Sekretarz Redakcji: mgr Grzegorz Zamorowski
Skład i Łamanie: dr inż. Mirosław Kordos

Rozprawy Naukowe nr 59

Adres Redakcji – Editorial Office:

WYDAWNICTWO NAUKOWE
AKADEMII TECHNICZNO-HUMANISTYCZNEJ
W BIELSKU-BIAŁEJ
43-309 Bielsko-Biała, ul. Willowa 2
tel./fax +48 33 8279 268

ISBN 978-83-66249-11-0
ISSN 1643-983X

Contents

Acknowledgements. The work was supported by the NCN (National Science Center, Poland) grant No. 017/01/X/ST6/00202 "Evolutionary Methods in Data Selection".

Streszczenie

Teza pracy. Redukcja rozmiaru zbioru danych pozwalająca na zwiększone możliwości doboru i przyśpieszenie uczenia modeli predykcyjnych, poprawę dokładności ich działania oraz ułatwienie analizy danych w zagadnieniach klasyfikacji i regresji może być skutecznie przeprowadzona poprzez dobrze opracowane algorytmy selekcji wektorów. Można to osiągnąć za pomocą trzech rodzin metod: opartych na podobieństwie, ewolucyjnych i wbudowanych w modele predykcyjne. Każda część książki poświęcona jest jednej rodzinie. W każdej rodzinie metod przedstawiono rozwiązania opracowane przez autora i zestawiono je z innymi istniejącymi rozwiązaniami oraz przedstawiono mocne i słabe punkty oraz obszary zastosowań poszczególnych metod.

Selekcja wektorów oparta na podobieństwie

Pierwszy rodział omawia wstępne przetwarzanie danych, które jest kluczowym krokiem w systemach eksploracji danych. Jest ono często ważniejsze niż wybór najlepszego modelu predykcji, ponieważ nawet najlepszy model nie może uzyskać dobrych wyników, jeśli uczy się przy użyciu niskiej jakości danych. Częścią wstępnego przetwarzania danych jest selekcja danych, która obejmuje selekcję cech i selekcję wektorów (instancji) i która znajduje zastosowanie praktyczne w szeregu problemów.

Celem selekcji wektorów jest zachowanie użytecznych informacji w danych oraz odrzucenie błędnych informacji, przy jednoczesnym zmniejszeniu rozmiaru danych poprzez wybranie optymalnego zestawu wektorów. Pozwala to na przyspieszenie uczenia modelu predykcyjnego i uzyskanie niższego błędu predykcji.

Zmniejszenie rozmiaru danych ułatwia również analizę właściwości danych przez ludzi, a także pozwala na ocenę oczekiwanej wydajności modeli predykcji. Innymi

słowy, poprzez selekcję wektorów, chcemy "skompresować informację". W tej pracy rozważamy selekcję wektorów w problemach zarówno klasyfikacji, jak i regresji.

W przypadku selekcji binarnej każdy wektor może być wybrany lub odrzucony. W przypadku ważenia można przypisać wektorom wartości rzeczywiste od 0 do 1, które odzwierciedlają ich znaczenie dla budowania modelu predykcyjnego. Następnie model uwzględnia udział poszczególnych wektorów w procesie uczenia się proporcjonalnie do przypisanych im wag.

W praktyce selekcja wektorów jest bardzo złożonym zagadnieniem i nie możemy powiedzieć, który wektor musi zostać odrzucony, nie biorąc pod uwagę, które inne wektory również zostały odrzucone. Dzieje się tak dlatego, że wynik zależy od całego zbioru wektorów, na którym są uczone modele predykcyjne, a zatem musimy rozważyć ten zbiór wektorów jako całość, co sprawia, że selekcja wektorów jest problemem NP-trudnym.

Drugi rozdział przedstawia przegląd popularnych algorytmów selekcji wektorów w zagadnieniach klasyfikacji a także zagadnienia redukcji kosztu obliczeniowego, użycia różnych modeli predykcyjnych wewnątrz algorytmów selekcji wektorów i dostosowania algorytmu k-NN GAS do danych etykietowanych. Nazywamy tu ten typ metod metodami selekcji wektorów opartymi na podobieństwie, ponieważ wykorzystują one w podejmowaniu decyzji podobieństwo wektorów albo w sensie odległości euklidesowej albo innych miar bliskiego położenia.

Trzeci rozdział omawia zagadnienie selekcji wektorów w problemach regresyjnych. Większość metod selekcji wektorów bazuje tu na analizie etykiet klas sąsiednich wektorów w celu określenia, czy dany wektor powinien zostać odrzucony czy też zaakceptowany. W przypadku regresji porównywanie etykiet nie jest możliwe, ponieważ wartości wyjściowe nie są nominalne, lecz ciągłe. W związku z tym zamiast etykiet należy ocenić inną wielkość. Przedstawiono dwa podejścia do tego problemu: metodę opartą na progach i metodę opartą na dyskretyzacji.

W metodzie opartej na wartości progowej proponowane podejście polega na zastosowaniu błędu modelu predykcyjnego (k-NN, sieci neuronowej, lub innego) przy przewidywaniu wartości danego wektora i porównaniu go z predefiniowanym progiem, aby określić, czy dany wektor powinien zostać zaakceptowany lub odrzucony:

jeżeli ($error > \alpha * std(k)$) to (.)

gdzie $std(k)$ jest odchyleniem standardowym wyjść k najbliższych sąsiadów danego wektora, natomiast (.) może oznaczać akceptację lub odrzucenie wektora, w zależności od danego algorytmu.

Aby poprawić wydajność tej metody, próg powinien zależeć od lokalnych właściwości zbioru danych. W zbiorach danych istnieją obszary o niższym i wyższym zróżnicowaniu danych. W obszarach bardziej jednorodnych nawet małe odchylenie

od wartości przewidzianej przez k-NN może oznaczać, że dany wektor stanowi szum i powinien zostać odrzucony przez T-ENN (Threshold-based ENN). W obszarach o większym zróżnicowaniu takie odchylenia mogą być normalne. W tym celu próg Θ ustanawiamy jako proporcjonalny do odchylenia standardowego wartości wyjściowych k najbliższych sąsiadów danego wektora.

To podejście można zastosować do dostosowania algorytmów ENN, CNN, DROP i innych bazujących na najbliższych sąsiadach w celu dostosowania ich do zagadnień regresji.

W metodzie opartej na dyskretyzacji podejście polega na bezpośrednim wykorzystaniu algorytmów selekcji wektorów dla zadań klasyfikacji. W tym celu zmienna wyjściowa jest najpierw dyskretyzowana, a zatem problem jest przekształcany w zadanie klasyfikacji wielo-etykietowej. Po zakończeniu selekcji wartość zmiennej wyjściowej jest przywracania do pierwotnej wartości.

To podejście pozwala na stosowanie zdecydowanej większości metod selekcji wektorów oryginalnie przeznaczonych dla zagadnień klasyfikacji do selekcji instancji dla zagadnień regresji. Dyskretyzacja jest kluczowym krokiem tej metody i ma decydujący wpływ na działanie algorytmu, ponieważ granica między klasami determinuje akceptację bądź odrzucenie wektorów. W większości przypadków dla 10 przedziałów o stałej szerokości uzyskuje się dość dobre wyniki.

Czwarty rozdział przedstawia zagadnienie ważenia wektorów. Często trudno jest określić optymalny próg odrzucenia, zarówno w przypadku selekcji wektorów opartej na progu do zadań regresji, jak w przypadku problemów klasyfikacyjnych. Choć tylko niektóre algorytmy selekcji wektorów dla klasyfikacji umożliwiają ustawienie takiego progu, to zawsze można to uzyskać stosując komitety selekcji wektorów z niedemokratycznego głosowaniem i w zależności od oczekiwanego stopnia kompresji regulować próg liczby członków komitetu, która musi za danym wektorem zagłosować.

Jest to szczególnie istotne, gdy selekcji wektorów używamy w roli filtra szumów. Zbyt liberalny próg nie odfiltrowuje całego szumu, podczas gdy zbyt rygorystyczny próg odfiltrowuje nie tylko szum, ale także pewne użyteczne dane. W takich sytuacjach możemy zastąpić ostry próg przypisaniem wektorom pewnej wagi z zakresu (0; 1). Im bardziej wektor wydaje się być odstającym, tym mniejsza waga jest mu przypisana. Każdy wektor jest następnie uwzględniany w uczeniu modelu proporcjonalnie do jego wagi. Na przykład w przypadku sieci neuronowej mnożąc błąd, który sieć daje w odpowiedzi na podanie danego wektora przez jego wagę, w przypadku k-NN oblicza się średnią ważoną k najbliższych sąsiadów dla regresji lub ważoną klasę większościową dla klasyfikacji.

Jako wagę istotności danego wektora proponuje się użyć iloczynu jego poszczególnych wag składowych (a przynajmniej tych z nich, które są wyznaczane): wagi atrybutów, wagi odległości, wagi gęstości danych, wagi odstających wartości.

Waga atrybutów to istotność poszczególnych atrybutów (cech), która może być wyznaczona za pomocą filtra cech. W przypadku algorytmu k-NN często to znacząco poprawia jego działanie. W przypadku sieci neuronowej mniej, bo sieć neuronowa już wewnętrznie dokonuje ważenie cech przez ustawianie odpowiednich wag połączeń wchodzących do neuronów.

Waga odległości to zwykła waga stosowana w ważonym k-NN polegająca na tym, że wektory znajdujące się dalej od danego wektora mają mniejszy wpływ na wyznaczenie jego wartości.

Waga gęstości została już omówiona przy wyznaczaniu współczynnika Θ.

Waga odstających wartości polega na tym, że im dany sąsiad interesującego nas wektora jest sam bardziej odstającym wektorem, tym jego wpływ jest mniejszy. Użycie tej wagi wymaga przeprowadzenia dwóch przebiegów procesu, gdzie za pierwszym razem wyznaczamy tą wagę na podstawie pozostałych trzech wag, a za drugim razem możemy już z niej skorzystać.

Piąty rozdział jest poświęcony komitetom algorytmów selekcji wektorów. Komitet jest modelem predykcyjnym złożonym z kilku prostych modeli pracujących równolegle. Ostateczną wartość predykcji uzyskuje się w najprostszym przypadku przez uśrednienie przewidywań jego modeli składowych w zagadnieniach regresji lub przez głosowanie w klasyfikacji. Ponieważ w tym kontekście selekcję wektorów można traktować albo jako zagadnienie regresji, gdy wektorom przypisujemy odpowiednie wagi, albo klasyfikacji w przypadku selekcji binarnej, można do tego celu zastosować znane z regresji i klasyfikacji rozwiązania komitetów, jak bagging, feature bagging i inne. Jednak są tu pewne ograniczenia wynikające stąd, że w zagadnieniach klasyfikacji wektorów prawie nigdy nie wiemy, jaka jest prawidłowa decyzja.

Komitety dają nam tu dwie zalety. Pierwszą jest to, że tak jak w klasyfikacji, komitet słabych modeli jest dobrym modelem na skutek tego, że poszczególne modele składowe mylą się w różnych przypadkach, a uśredniony wynik ich głosowania jest znacznie częściej poprawny. Drugą jest to, że taki komitet wcale nie musi głosować demokratycznie. Np. jeśli komitet składa się z 30 algorytmów selekcji wektorów to można przyjąć, że wektor zostaje wybrany, jeśli głosowało za nim 15 algorytmów, ale również można przyjąć, że wtedy, gdy tylko 10, lub gdy co najmniej 20 albo 25. Daje to możliwość większego kształtowania rozwiązań i uzyskania całego frontu Pareto poprzez różne proporcje głosowania. To jest szczególnie istotne w przypadku danych klasyfikacyjnych, gdzie większość algorytmów selekcji wektorów nie posiada możliwości kontroli progu, tak jak można to zrobić regulując Θ w selekcji wektorów w zagadnieniach regresji. Przedstawione rozważania są zobrazowane przykładami otrzymanych frontów Pareto.

Szósty rozdział pierwszej części omawia łączną selekcję cech i wektorów. Ponieważ wybór cech i wektorów wpływa na siebie nawzajem, należy ustalić właściwą kolejność selekcji tych dwóch wielkości, aby uzyskać najlepsze wyniki. Ogólna za-

sada jest taka, że w selekcji danych należy najpierw usunąć szum, a potem dokonywać kompresji (usunięcia wielkości zbędnych, redundantnych).

W zadaniach klasyfikacyjnych większość wektorów można usunąć, ponieważ są one nieistotne, ponieważ znajdują się daleko od granic decyzyjnych. Pozostawienie ich w żaden sposób nie wpływa na dokładność przewidywania. W przeciwieństwie do tego znaczny procent cech często stanowi szum, a ich usunięcie poprawia dokładność klasyfikacji. Podczas selekcji wektorów musimy precyzyjnie określić granice klas, dlatego pożądany jest zestaw cech z dużą zdolnością przewidywania. Na podstawie tej analizy można zaproponować, że selekcję cech najczęściej należy dokonać przed selekcją wektorów. Rzeczywiście nasze eksperymenty dowiodły, że jest to najlepsza opcja. W eksperymentach przetestowaliśmy również metodę iteracyjną, gdzie za każdą iteracją usuwaliśmy tylko jedną funkcję i kilka instancji. To jednak nie zaowocowało, a w większości przypadków nawet gorszym wynikiem, a rozwiązanie było bardziej złożone i czasochłonne.

Sytuacja się zmienia, gdy szum jest zawarty bardziej w wektorach niż w cechach. W takim przypadku należy najpierw dokonać selekcji wektorów, a następnie cech. Przy podobnym rozkładzie najlepiej sprawdza się metoda iteracyjna, gdzie w każdej kolejnej iteracji usuwa się najbardziej odstające cechy i wektory.

Metody ewolucyjne selekcji wektorów

Siódmy rozdział książki przedstawia w skrócie zasadę działania algorytmów genetycznych jedno i wielokryterialnych na przykładzie algorytmu NSGA-II.

Algorytmy ewolucyjne nie przyjmują żadnych założeń dotyczących właściwości rozwiązania, ale weryfikują iteracyjnie dużą liczbę różnych możliwych rozwiązań w sposób inteligentny, aby zminimalizować przestrzeń poszukiwań. Skutkuje to często lepszymi rozwiązaniami lub lepszym (niższym) frontem Pareto w przypadku wielu rozwiązań. Z drugiej strony jest to zwykle osiągane kosztem znacznie wyższych kosztów obliczeniowych. Z tego powodu w tej pracy zwracamy szczególną uwagę na ograniczenie kosztów obliczeniowych tak dalece, jak to możliwe, uzyskując porównywalne nakłady obliczeniowe z metodami selekcji wektorów opartymi na podobieństwie, nie rzadko nawet niższe.

Ósmy rozdział poświęcony jest wykorzystaniu jednokryterialnych algorytmów genetycznych do selekcji wektorów. Zaletą metody selekcji wektorów opartej na algorytmach ewolucyjnych jest to, że algorytm ewolucyjny ocenia jakość prognozowania dla całych podzbiorów wybranych wektorów i nie musimy jednoznacznie definiować relacji wektorów do ich sąsiadów, aby zdecydować o jej wyborze lub odrzuceniu. A to w metodach opartych na podobieństwie stanowiło istotny problem, zwłaszcza w zagadnieniach regresji, które są bardziej złożone w tym względzie.

Kolejną zaletą metod ewolucyjnych jest możliwość uzyskania rozwiązań z niższym błędem predykcji i większą redukcją danych (kompresją), czyli lepiej spełniających oba kryteria oceny selekcji danych.

W przypadku algorytmów jednokryterialnych funkcję dopasowania (fitness) można wyrazić w postaci:

$$fitness = \alpha \cdot compression + (1 - \alpha) \cdot accuracy \tag{1.1}$$

lub innej podobnej postaci, gdzie występują dwa kryteria, których waga jest balansowana parametrem α.

Stosujemy najprostsze możliwe kodowanie problemu: każdy osobnik reprezentuje jeden zbiór danych, a każda pozycja w chromosomie jeden wektor. Wartość zerowa na tej pozycji oznacza, że nie został on wybrany, a wartość większa od zera jego wagę. W przypadku selekcji binarnej wektorów dopuszczalnymi wartościami będzie tylko 0 i 1, zaś w przypadku ważenia wektorów liczba rzeczywista od 0 do 1.

Uzasadnieniem wyboru k-NN jako wewnętrznego algorytmu oceny jest szybkość tego podejścia. Dzieje się tak dlatego, że pełny algorytm k-NN musi zostać wykonany tylko raz przed rozpoczęciem optymalizacji. W przypadku innych algorytmów predykcji byłoby to albo niemożliwe, albo znacznie bardziej złożone, a przez to mniej wydajne. Załóżmy, że w populacji jest 100 osobników i że optymalizacja wymaga 30 epok. W tym przypadku wartość funkcji dopasowania musi być obliczona 3000 razy. Uczenie każdego modelu 3000 razy byłoby obliczeniowo bardzo kosztowne.

Chociaż nasze poprzednie eksperymenty pokazały, że najlepsze wyniki pod względem balansu kompresja-klasyfikacja lub kompresja-$rmse$ można zwykle uzyskać, jeśli wewnętrzny model oceny jest tym samym algorytmem, co końcowy predyktor, to w tej pracy poświęcamy tę niewielką poprawę, aby skrócić optymalizację procesu, często nawet o dwa lub trzy rzędy wielkości. Jednakże, jak pokażemy, kiedy końcowym predyktorem jest sieć neuronowa MLP, do oceny wewnętrznej używamy k-NN z odpowiednio dobranymi parametrami, m. in. poprzez system wag omówiony w pierwszej części pracy, co sprawia, że jego przewidywania są możliwie zbliżone do przewidywania sieci neuronowej. W ten sposób uzyskujemy lepsze wyniki, przy jednoczesnym zachowaniu krótkiego czasu procesu selekcji wektorów.

W przypadku algorytmu k-NN obliczamy macierz odległości pomiędzy każdą parą instancji w zbiorze treningowym i następnie sortujemy ją w jednym z wymiarów, przechowując tylko wartości wyjściowe kolejno uporządkowanych sąsiadów oraz numery ich w zbiorze. Następnie obliczamy średnią (w regresji) lub klasę większościową (w klasyfikacji) z k kolejnych sąsiadów, które zostały wybrane w danym osobniku.

Dziewiąty rozdział omawia zastosowanie wielkokryterialnych algorytmów genetycznych do selekcji wektorów, a w szczególności algorytmu NSGA-II i jego modyfikacji. W przypadku optymalizacji wielokryterialnej, kluczową zaletą jest to, że otrzymujemy pulę rozwiązań zlokalizowanych na froncie Pareto, gdzie każde z nich

jest najlepsze dla określonego balansu kompresji jakości predykcji (dokładności klasyfikacji lub błędu średniokwadratowego) i możemy wybrać jedno z tych rozwiązań.

Najlepsze rozwiązania otrzymaliśmy bazując na wielokryterialnym algorytmie genetycznym NSGA-II i czyniąc szereg jego dostosowań do zagadnień selekcji i ważenia wektorów tak w zagadnieniach klasyfikacji, jak i regresji.

W rozdziale jest szczegółowo omówiony szereg mechanizmów mających na celu zarówno dalsze przyśpieszenie procesu selekcji wektorów, jak i poprawę jakości otrzymywanych rozwiązań, popartych przykładami i graficzną reprezentacją otrzymanych wyników. Przedstawione są także porównania z metodami opartymi na podobieństwie oraz graficzne przykłady otrzymanych frontów Pareto dla różnych parametrów. Jest to najdłuższy rozdział.

Dziesiąty rozdział przedstawia dodatkowe opcje pozwalające uzyskać lepsze wyniki oraz krótszy czas obliczeń przy dużych zbiorach danych. Partycjonowanie danych może być używane w przypadku selekcji wektorów w dwóch celach: przyspieszenia obliczeń i poprawy wyników. Pierwszy cel jest ważny tak w ewolucyjnej, jak w opartej na podobieństwie selekcji wektorów. Drugi cel jest jednak ważniejszy w ewolucyjnym procesie, ponieważ pozwala uruchomić optymalizację w poszczególnych obszarach zbioru danych pod kątem optymalnej liczby iteracji. Jest to ważne, ponieważ optymalna liczba iteracji różni się w zbiorze danych treningowych. Jeśli optymalizacja zostanie przeprowadzona zbyt krótko, optymalne rozwiązanie nie zostanie znalezione. Jeśli zbyt długo, to prawdopodobnie wystąpi nadmierne dopasowanie; wyniki na zbiorze treningowym będą się poprawiać, a wyniki na zbiorze testowym zaczną stopniowo spadać. Inną kwestią jest to, że algorytmy genetyczne często są mniej wydajne, jeśli chromosom jest zbyt długi (dziesięć tysięcy pozycji lub więcej), to znaczy wymagają więcej iteracji.

Partycjonowanie danych jest bardziej efektywnym rozwiązaniem w problemach regresji. Dzieje się tak dlatego, że w przypadku problemów regresyjnych zmiany są równo rozłożone w przestrzeni danych. W przypadku problemów klasyfikacyjnych ważne jest, aby zachować granice decyzyjne i podczas partycjonowania przestrzeni danych musimy zachować szczególną ostrożność, aby nie dzielić danych wzdłuż granic decyzyjnych, ponieważ selekcja wektorów po prostu nie będzie działać z takimi partycjami.

Co więcej, nie musimy obliczać dla każdego wektora całej macierzy odległości używanej przez k-NN, ale tylko odległości do kilku jego najbliższych sąsiadów. Możemy tu użyć grupowania danych (klasteringu), np. metody k-średnich, aby zgrupować dane w kilka klastrów, a następnie obliczyć odległości tylko w obrębie poszczególnych klastrów lub w wersji dokładniejsze także klastrów sąsiednich. Bowiem klastering k-średnich ma w praktyce złożoność około $O(k \cdot n)$, gdzie k jest liczbą klastrów. Następnie złożoność obliczania wszystkich macierzy odległości tylko w obrębie poszczególnych klastrów jest już w przybliżeniu liniowa.

Jednym z ostatecznych celów selekcji wektorów jest zminimalizowanie $rmse$ na zestawie testowym, ale celem używanym bezpośrednio podczas procesu wyboru instancji jest $rmse$ na zbiorze treningowym, ponieważ zestaw testów jest nieznany podczas optymalizacji. Wymusza to wczesne zatrzymanie (early stopping), zanim nastąpi nadmierne dopasowanie do danych. W przypadku dużych zbiorów danych występuje problem z określeniem optymalnego punktu zatrzymania, ponieważ w niektórych obszarach zbioru danych optymalizacja zbiega się szybciej niż w innych. Goldberg napisał [1], że algorytmy ewolucyjne działają "przez budowanie krótkich, niskich rzędów i wysoce dopasowanych schematów (bloków), które są rekombinowane i ponownie próbkowane w celu utworzenia łańcuchów o potencjalnie wyższej sprawności". W ten sposób dopasowanie może już się rozpocząć w takim bloku, podczas gdy w innych częściach chromosomu potrzeba jeszcze więcej iteracji, aby zbliżyć się do optymalnego punktu.

W pracy omawiane jest również zagadnienie wykorzystania kilka frontów Pareto w optymalizacji wielokryterialnej celem osiągnięcia bardziej zróżnicowanej puli rozwiązań i redukcji możliwości przeuczenia modelu.

Jedenasty rozdział poświęcony jest skracaniu czasu obliczeń i optymalizacji selekcji wektorów z wykorzystaniem algorytmów genetycznych. Omawiane są szczegółowo zagadnienia doboru optymalnych metod inicjalizacji danych, metod krzyżowania wielopunktowego, doboru optymalnej wielkości populacji, optymalizacji wewnętrznych funkcji kryterialnych algorytmu genetycznego, możliwości dynamicznego skrócenia długości chromosomu podczas obliczeń oraz innych rozwiązań mających na celu poprawę jakości otrzymywanych zbiorów oraz skrócenie czasu obliczeń.

Omawiana jest także szczegółowo złożoność obliczeniowa zastosowanego rozwiązania oraz poszczególne jej składowe. Metoda ma liniową złożoność obliczeniową.

W pracy przeprowadzono również porównania eksperymentalne wybranych (możliwie najlepszych) metod selekcji wektorów wykorzystujących różne metody działania.

Dwunasty rozdział przedstawia temat jednoczesnej selekcji cech i wektorów z wykorzystaniem algorytmów ewolucyjnych, w tym przez metody sekwencyjne, poprzez podział chromosomu na sekcję cech i wektorów, oraz z użyciem algorytmów koewolucyjnych, utrzymujących populację cech i populację wektorów.

Trzynasty rozdział poświęcony jest selekcji wektorów w zagadnieniach wieloetykietowych, gdzie należy dokonać selekcji wektorów tak, aby wybrany zbiór minimalizował błąd predykcji jednocześnie dla każdego wyjścia. Przedstawione są różne możliwości optymalizacji w zagadnieniach wieloetyketowych, jak single target model, model chain, multi-target stacking, ensemble of model chains, które wykorzystują wzajemne powiązania między wyjściami. Przykładowo łańcuch modeli (model chain) zaczyna predykcję od pierwszego wyjścia korzystając tylko z wejść, a następnie dodaje to przewidziane wyjście jako kolejne wejście (opcjonalnie z inną wagą) do

przewidywania kolejnego wyjścia, itd. Przedstawione metody wykorzystują algorytm NSGA-II do przeszukiwania przestrzeni rozwiązań, wraz z większością usprawnień omówionych w poprzednich rozdziałach. Wyniki uzyskane na benchmarkowych zbiorach regresyjnych wielowyjściowych pozwoliły na znaczne ograniczenie wielkości zbioru przy jednoczesnym zmniejszeniu błędu predykcji.

Selekcja wektorów wbudowana w sieci neuronowe

Czternasty rozdział przedstawia w skrócie zasadę działania sieci neuronowych typu MLP oraz dwa algorytmy ich uczenia używane w tej pracy: Rprop i VSS opracowany przez autora niniejszej książki. Algorytmy te są wykorzystywane w dalszych rozdziałach w uczeniu sieci neuronowych oraz w selekcji wektorów za pomocą sieci neuronowych. Zaletą algorytmu VSS jest to, że nie wymaga on ciągłych ani różniczkowalnych funkcji odpowiedzi sieci neuronowej, a zatem dobrze nadaje się do zastosowania w uczeniu sieci neuronowej nieczułej na szum.

Piętnasty rozdział omawia metody selekcji wektorów wbudowane w uczenie sieci neuronowych, koncentrując się na selekcji i ważeniu wektorów celem redukcji szumów.

Sieci neuronowe typu perceptrona wielowarstwowego (MLP) są uczone przez minimalizowanie funkcji błędu na zbiorze treningowym. Taki sposób uczenia sieci jest silnie uzależniony od jakości danych uczących. Algorytmy uczenia sieci próbują dopasować punkty danych tak blisko, jak to możliwe. Jest to ewidentnie oczywiste, że wartości odstające i błędne będą wpływać na końcową jakość predykcji sieci, prowadząc do niewłaściwego odwzorowania przestrzeni wejściowej na wyjście. Najczęściej stosowana miara błędu, błąd średnio-kwadratowy (rmse), może być uważana za optymalną tylko dla czystych danych treningowych lub danych zanieczyszczonych przez co najwyżej błędy generowane z rozkładu Gaussa o zerowej średniej.

Aby rozwiązać ten problem, zaproponowano kilka metod, głównie opartych na zmodyfikowanych miarach błędu. Zastępują one kryterium $rmse$ nową funkcją błędu, opartą na koncepcji tzw. metod odpornych na błędy w danych. Omówiono w pracy szereg takich funkcji statycznych (ILMedS, LTA, MedSum i inne) oraz funkcje dynamiczne, czyli takie, które się zmieniają podczas uczenia sieci neuronowych. Charakterystyczną cechą funkcji z obu tych grup jest to, że słabiej reagują na duże błędy, tj. duży błąd sieci przepuszczony przez taką funkcję staje się mniejszym błędem w ten sposób w mniejszym stopniu wpływając na wynik końcowy uczenia sieci.

Idea przedstawionych dynamicznych funkcji błędów polega na płynnym zmniejszeniu wpływu wartości odstających na trening sieciowy. Zatem, jeśli różnica pomiędzy rzeczywistą a pożądaną odpowiedzią sieci jest mała, to błąd rośnie wraz z tą różnicą. Gdy różnica dalej rośnie, błąd przestaje rosnąć, a ostatecznie zaczyna zmniejszać

się z dalszym wzrostem różnicy, gdyż różnica jest przypuszczalnie związana z wektorem odstającym, którego wpływ na uczenie sieci należy zminimalizować dokonując w ten sposób jego odrzucenia. Jednak funkcje takie muszą być wprowadzane stopniowo, ponieważ na początku treningu wagi sieci są losowe, a wysoka wartość różnicy między wartością oczekiwaną a uzyskaną nie musi wskazywać na wektor odstający.

Szesnasty rozdział omawia połączenie metod selekcji wektorów wbudowanych w sieci neuronowe z metodami opartymi na podobieństwie oraz zawiera opracowania przeprowadzonych eksperymentów.

Siedemnasty rozdział omawia połączenie selekcji cech wbudowanej w sieci neuronowe z selekcją wektorów. Selekcję cech z wykorzystaniem sieci neuronowych można wykonać na kilka sposobów. Dwa podstawowe podejścia to analiza wag, w tym metod przycinania i perturbacji danych wejściowych. W analizie perturbacyjnej zastępujemy wartości danej cechy losowymi wartościami w wektorach testowych i obserwujemy, jak wpływa to na dokładność sieci. W analizie wagi zakładamy, że mniej ważne cechy wygenerują mniejsze bezwzględne wartości wag i możemy odrzucić cechy o niższej ważonej sumie wag w warstwie wejściowej. Wagi można również wymusić na małych wartościach poprzez dodatkowy człon regularyzacyjny. Analiza wag została wykorzystana w naszych eksperymentach. Natomiast bardziej złożona metoda uwzględnia również pochodne.

Następnie omówiona jest redukcja wektorów nadmiarowych. Ze względu na nieliniowe funkcje transferu odpowiedzi sieci na poszczególne wektory zależą od aktualnego położenia punktu na funkcji transferu, a w zagadnieniach klasyfikacji na końcu procesu uczenia sieci pozycja ta jest przeważnie w nasyconym obszarze, dlatego należy dokonać pewnych dostosowań tak, aby zapobiec zbyt dobremu nauczeniu się sieci przed dokonaniem selekcji wektorów, co pozwoli na ich zlokalizowanie i wyeliminowanie.

W rozdziale tym przeprowadzone są też porównania wyników otrzymanych tymi metodami i w oparciu o metody oparte na podobieństwie oraz w oparciu o połączenie tych dwóch grup metod.

Osiemnasty rozdział przedstawia zagadnienie ekstrakcji reguł logicznych z sieci neuronowych i przegląd przykładowych metod. Znaczna część rozdziału poświęcona jest specjalnej strukturze i dopasowanemu do niej algorytmowi uczenia sieci neuronowej opracowanej przez autora, która umożliwia jednoczesną predykcję, selekcję cech i wektorów oraz dekompozycyjną ekstrakcję reguł logicznych z danych.

Główny problem selekcji cech z użyciem sieci neuronowych, a jeszcze bardziej ekstrakcje reguł logicznych jaki występuje przy klasycznej sieci neuronowej typu MLP jest bardzo złożona propagacja sygnału przez sieć, ponieważ szczególne wagi ukrytych neuronów są wspólne dla każdego neuronu wyjściowego reprezentującego każdą klasę. Przedstawione rozwiązanie używa dedykowanych ukrytych neuronów

dla każdej klasy w zagadnieniach klasyfikacji dla każdego przedziału wartości wyjściowej w problemach regresji. Pozwala to znacznie uprościć analizę, a także sprawia, że sieć jest łatwiejsza do nauczenia, ponieważ problem z klasyfikacji wieloetykietowej zostaje sprowadzony do wielu problemów jednoklasowych. Jeśli dane są ciągłe, muszą zostać zdyskretyzowane przed uczeniem sieci. Neurony mają sigmoidalną funkcję transferu, której nachylenie stopniowo się zwiększa podczas uczenia sieci, aż do funkcji skokowej. W połączeniu z członem regularyzacyjnym wymuszającym ostateczne przyjęcie przez wagi tylko jednej z trzech możliwych wartości (-1, 0, 1) umożliwia to prostą ekstrakcję reguł logicznych. Dodatkowo celem zwiększenia dokładności, zwłaszcza w zagadnieniach regresji dopuszcza się możliwość dodania prostego członu reguły skośnej w obrębie poszczególnych neuronów wyjściowych.

Istnieją dwa mechanizmy selekcji wektorów celem redukcji szumu: mechanizm omawiany już w tradycyjnych sieciach neuronowych, oraz mechanizm oparty na eliminacji wektorów nie reprezentowanych przez żaden neuron warstwy ukrytej.

Podsumowanie zestawia możliwości selekcji wektorów oraz wskazuje możliwe dopasowania optymalnych rozwiązań w zależności od problemu oraz oczekiwań użytkownika.

Do najważniejszych metod opracowanych przez autora można zaliczyć:

1. algorytmy selekcji wektorów dla zagadnień regresji
2. schematy ważenia wektorów
3. modele komitetów algorytmów selekcji wektorów
4. jednokryterialne metody ewolucyjne selekcji wektorów
5. wielokryterialne metody ewolucyjne selekcji wektorów w zagadnieniach klasyfikacji i regresji danych jedno- i wieloetykietowych
6. aglorytmy selekcji wektorów celem redukcji szumu wbudowane w sieci neuronowe
7. algorytmy selekcji wektorów celem redukcji rozmiaru zbioru danych wbudowane w sieci neuronowe
8. algorytmy łącznej selekcji cech, wektorów i ekstrakcji reguł logicznych w zagadnieniach klasyfikacji i regresji wbudowane w sieci neuronowe
9. połączenie metod selekcji wektorów z różnych grup oraz selekcji wektorów z selekcją cech
10. inne usprawnienia w metodach selekcji wektorów.

About the Book

The thesis of the work is that instance selection can be effectively used to improve prediction accuracy by noise reduction and to reduce the data size, however the solutions must be properly selected and tuned to the specific problem. It can be achieved with three families of methods: similarity-based, evolutionary and embedded. Each part of the book is dedicated to one family. In each method family the solutions developed by the author are presented, compared with other existing solutions and the strong and weak points and areas of application of the methods are presented.

The purpose of data preprocessing, which includes instance selection, is to improve the data quality, as in machine learning the accuracy of the predictive models is limited by the quality of the training data. Instance selection removes noisy and unnecessary data. This improves the model prediction, accelerates its training and makes the data easier to analyze and interpret. Variants of instance selection are instance weighting and instance generation. In instance weighting each instance is assigned a weight expressing its importance, which will be used in the predictive model training. In instance generation new, more representative instances are generated to replace the original ones. The book presents the newest and most effective solutions in three groups of instance selection approaches and joint instance and feature selection, especially those developed by the author and his research group and discusses their complexity, efficiency and usage areas.

In the first part of the book we discuss a group of instance selection algorithms, which we call "similarity-based methods", because they use some similarity measures between the instances, as the Euclidean distance or relative position on the Voronoi cells. We will start from presenting this group including adaptation of the instance selection algorithms to regression problems, optimization of the instance weights reflecting the attribute, neighbor, distance, density and gradient properties, enhancements of ensemble methods in instance selection and concurrent instance and feature

selection. The advantage of this group is speed, simplicity of implementation and possibility to get explanation why particular instances were selected. Also the correlation between the data properties, performance of the predictive model and outcome of instance selection algorithms will be discussed.

In the second part evolutionary instance selection methods will be discussed. In genetic algorithms typically each instance is encoded into one position in the chromosome. In a single objective optimization we set a parameter to balance the two objectives: data reduction and prediction error. In multi-objective optimization we obtain a set of the best solutions on the Pareto front. As the instance selection is performed on the training set and the final purpose it to obtain good accuracy on the test set, the process is prone to over-fitting and solutions to this problem will be presented, as new population initialization schema, adjusted mutation operators or dataset partitioning. New methods of improving the results and accelerating the process will be discussed, as proper selection of the inner evaluation model, performing the whole training algorithm only once and then only very little additional calculations while evaluating the fitness function and optimization of particular parameters of the evolutionary algorithm. Also joint evolutionary instance and feature selection and the relation between ensembles and Pareto front solutions will be discussed. The advantage of this group is that usually higher prediction accuracy and stronger data reduction can be obtained.

In the third part we present how instance selection embedded into predictive models can be incorporated into the model learning process. The book focuses on neural network models. The basic idea here is that the instance properties are reflected by the model response to presenting that instance. In general high error indicates outliers and very low error may indicate an unnecessary instance. However, that depends on the network training phase and the considered problem. Thus the network training process should be appropriately modified. Also special network architectures and learning algorithms specially designed for instance selection and logical rule extraction will be introduced, where the trade-off between the data reduction, interpretation simplicity and accuracy can be addressed, e.g. by implementing incremental network structures. Methods of instance assessing, weighting and removing will be discussed, including joint instance and feature selection. The advantage of this group is that a separate step of instance selection is not required, the obtained results are optimized for that particular predictive model and to some extent we can also get explanation why particular instances were selected. Finally a comparison of the approaches and usage recommendations are provided.

The main contributions of the author include development of:

1. instance selection algorithms for regression tasks
2. instance weighting schemes
3. ensemble models of instance selection algorithms
4. single-objective methods of evolutionary instance selection
5. multi-objective methods of evolutionary instance selection for classification and regression tasks of single and multi-label data
6. a number of improvements in evolutionary instance selection methods
7. instance selection methods for noise reduction embedded in neural networks
8. instance selection methods for data condensation embedded into neural networks
9. methods for joint selection of features, instances and extraction of logic rules in classification and regression tasks embedded into neural networks
10. methodology for combining methods of instance selecting from different groups and combining instance selection with feature selection.

Part I
Instance Selection with Similarity-based Methods

Chapter 1
Introduction to Instance Selection

Abstract In this chapter we introduce the concept of data selection, which consists of feature selection and instance selection. We define the different purposes of instance selection, as data condensation, noise reduction and different approaches to these tasks. We also present the datasets used in experiments described in this book.

1.1 Data: Features and Instances

Classification and regression tasks are very common problems in our everyday life. As a results they are also the most frequent problems in machine learning, which is aimed to help us make the correct decision or to make this decision for us. In a classification problem an object is assigned to one of the predefined categories, called classes, based on the object properties. In a regression problem an object is assigned a real value based on the object properties.

When machine learning is used to predict the object class or real value, it is done based on the object similarity to the known objects. In order to accomplish this, various machine learning models analyze the dataset of known objects and using some algorithms and heuristics determine the most probable class or real value of the object. Sample datasets organized in the matrix form are shown in Tables 1.1 and 1.2. Each object in the dataset is known as an instance or vector and is represented by one row, while each column represents one feature.

Some instances can be either redundant or may contain erroneous values. E.g. in Table 1.1 $instance4$ and $instance5$ are redundant, so one of them can be removed from the dataset. $instance6$ is probably wrongly annotated, as its features indicate that it should belong to class A, while it belongs to class B. In order to improve the data quality also $instance6$ should be removed. In Table 1.2 $instance4$ and $instance5$ contain very similar values and probably they are redundant, but this is not clear in

Table 1.1. Dataset format for classification tasks.

	feature_f1	**feature_f2**	**class**
instance1	209	6	A
instance2	307	8	A
instance3	240	9	A
instance4	292	7	A
instance5	292	7	A
instance6	237	8	B
instance7	840	26	B
instance8	892	23	B
instance9	830	25	B
instance10	890	27	B

Table 1.2. Dataset format for regression tasks.

	feature_f1	**feature_f2**	**output**
instance1	209	6	0.120
instance2	245	7	0.140
instance3	305	8	0.160
instance4	352	9	0.180
instance5	354	9	0.181
instance6	367	10	0.300
instance7	804	26	0.375
instance8	848	27	0.400
instance9	898	28	0.425
instance10	932	29	0.450

this case, as they are not identical. Can one of them get removed? To answer this question we have to analyze how this removal will influence the prediction model that learns from this dataset. Also $instance6$ seems to be an outlier - its features indicate that the output should be about 0.185, while it is 0.300. But maybe it is not, maybe the data does not change linearly and there is some sudden step between $instance5$ and $instance6$? So should $instance6$ get removed? Or maybe it should be treated by the learning model with some uncertainty? As we can see the question of how some instances should be treated is not always obvious and in this book we will discuss various approaches to that topic.

Features, as well as instances can be either redundant or can bring no useful information or even can bring some wrong information.

1.2 Purpose and Idea of Instance Selection

Before the prediction model can learn the data representation, first the data must be collected and then prepared appropriately. This stage is called data preprocessing and it is a crucial step in data mining systems. It is frequently more important than choice of the best prediction model, as even the best model cannot obtain good results if it learns using poor quality data [2]. A part of data preprocessing, on which we focus in this book is data selection that comprises feature selection and instance selection.

The purpose of both feature selection and instance selection is to preserve useful information stored in the data and reject the erroneous and redundant information, by selecting an optimal set of features and instances. This allows to accelerate the predictive model training, to obtain a lower prediction error and to make the data analysis easier [3].

While redundant instances do not make usually any direct harm to the accuracy of the predictive model, except the cases, where because of too big data size a limited number of solutions can be examined, outliers can easily decrease prediction accuracy of the model. An outlier can be defined as an observation numerically distant from the majority of the data. Such a pattern can be a point that is close to its neighbors in the input space, but far from them in the output space (different class or much different value in the case of regression) or that is far from any points as well in the input as in the output space. Outliers may be generated as measurement artifacts, rounding errors, human mistakes, long-tailed noise distribution, etc. According to [4], the quantity of outliers ranges from 1% to 10% in typical raw data. However the percentage of outliers is hard to predict in the real data and according to our experience the amount of outliers in some industrial data can be even higher than 10%. Detecting such points is not trivial, moreover, sometimes it cannot be clearly stated if a given point is an outlier or not and rather some degree of outlierness than a crisp decision is preferred. In general, while building a data-driven model we do not intend to disregard such a point but only weaken its influence on the model parameters. Reducing the data size also makes it easier to analyze the properties of the data by humans, as well as can allow assessing the expected performance of the prediction models [5]. In other words by instance selection we want to "compress and improve the information".

Data selection, including instance selection finds practical application in a range of problems, where the data size can be reduced. For example, it can be applied to the datasets considered in this book, describing real-world problems from various domains. Also the author took part in several practical implementations of instance selection in the industry. One was an artificial intelligence-based system for controlling steel production in electric arc process, where there was a lot of data from the previous processes (such as the amount of energy, of different chemical compounds, etc.). Another one was in the electronics industry in a system for predicting the performance of the electronic appliances (power inverters and others) where the amount of data describing the parameters and behavior of the appliances was also very large. In

both cases, there were regression problems with many redundant and erroneous data and instance selection was very useful to enable efficient further processing of the data.

The first difference between instance selection for classification and regression tasks is that in classification it is enough to determine the class boundaries (the black line in Fig. 1.1 left), and to select only the instances needed to determine the boundaries [6]. The remaining instances (in the gray-green area) can be removed. However, before removing them the noisy instances, which do not match their neighbor class must be removed first in order not to introduce the false classification boundaries.

In case of instance selection in regression tasks we also need to remove the noisy instances, which do not match their neighbors (instances A and B in Fig. 1.1 right) and then we can remove the instances that are very close to some other instances in the input and the output space (instances C and D in Fig. 1.1 right). However, the reduction cannot be so strong as in classification problems, where we need only the class boundaries, because in regression each region in the data space is important.

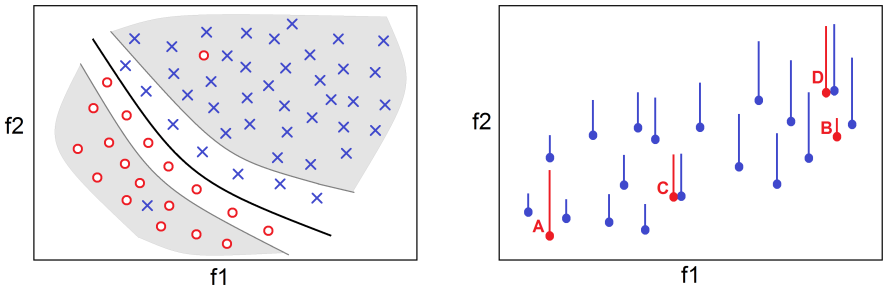


Fig. 1.1. Instance selection in classification (left) and in regression (right). The axes represent the features $f1$ and $f2$. In classification red circle and blue cross represent points of two different classes. In regression the height of the vertical line represents the output value of an instance and the circle shows its location in the input space.

Obviously, we always want to select the most representative instances, so that the reduced set contains as much useful information and as low noise as possible. However, in practice instance selection is not so simple as in Fig. 1.1, where there are only two attributes and a few instances.

Although the binary instance selection process can be considered a two class classification process, there are two fundamental differences between classification and instance selection. First, in instance selection we do not know the true label (selected or rejected) and second the decision if a particular instance should be selected strongly depends on the selection of its neighbors, as the whole combination of instances constitute the optimal training set. This makes instance selection an NP-hard problem and

it is impossible to examine all the combinations of selected sets in order to find the best one. The number of possible different sets of selected instances nk is given by the following formula:

$$nk = \sum_{k=1}^{N} \binom{N}{k} = \sum_{k=1}^{N} \frac{N!}{k!(N-k)!} = 2^N - 1 \approx 10^{0.3N} \tag{1.1}$$

For example, if there are N=1000 instances in the original dataset, we can obtain about nk=10e+301 selected subsets. This is clearly seen that training the model on 10e+301 different subsets and selecting the best solution is impossible. Even if we evaluate 1e+9 solutions per second, it would take about 3e+284 years. In the case of instance weighting, which will be discussed in the subsequent chapters, the number of possible combinations get even much higher, depending on the weighting scheme. Thus it is clearly evident that there is a need to design some instance selection algorithms that will be more intelligent than just brute force.

For that reason, many methods of instance selection have been developed, including some approaches, which jointly consider instance and featured selection. In this book we will present three groups of the methods:

- instance selection with similarity-based methods
- instance selection with evolutionary methods
- instance selection embedded into learning models

In each of the groups the purpose of instance selection can be:

- noise removal (the methods are called noise filters)
- data size reduction (the methods are called condensation methods)
- noise removal and data size reduction

We will discuss the idea of each group, the problems that must be solved and implementations of some instance selection methods. A significant part of the book is based on the author's research, thus the instance selection methods presented here in detail will be those developed by us, but also several methods of other authors will be shortly presented to make the work comprehensive by providing as well the background as the state of the art solutions.

A special kind of instance selection are instance generation and instance weighting. In binary instance selection, each vector (instance) can be either selected or rejected. In instance weighting, the instances can be assigned real value weights between 0 and 1, which reflect the instance importance for building the predictive model. Then the model includes the contribution of particular instances in the learning process proportionally to the weights assigned to them [7].

$$Weight(x) = F(Properties(neighbors(x)), Model(x, neighbors(x))) \tag{1.2}$$

where $Weight(x)$ is a value that indicates the importance of the vector x (how much vector x is required or desired), Properties of the neighbors of x can express local diversity of the data (e.g. standard deviation) or the trends in the data variability, $Model(x, neighbors(x))$ is some predictor (classifier) that learns on $neighbors(x)$ and predicts x; that can be k-NN, neural network, etc. However, in most of the classical instance selection methods, $Model(.)$ is based on some distance or neighborhood concept, as it is simple and computationally efficient.

Then we set some threshold, and the instances with their weight below the threshold get rejected. However, in instance weighting they still can be kept in the training dataset and their influence on the model will be multiplied by their weights. For example if the model is a neural network, then the error the network makes on each instance is multiplied by the instance weight, if the model is k-NN, then a weighed voting takes place.

For example in a noise filter method, k-NN can be used to predict the class of each vector. If the real class of the vector is different than the predicted class then the vector can be considered and outlier and will be marked for removal. In case of instance weighting, the weight can be proportional to the percentage of the nearest neighbors that belong to the same class as the query instance. Thus the vectors, which differ much from their neighbors will not be removed, but will have lower influence on learning the predictive model.

In regression problems there are no classes, but still we can use some similarity measures between instances. In a similar way as in classification problems, in the case of noise filters the instances that differ too much from their neighbors will be rejected in instance selection or assigned a very small weights in instance weighting.

Condensation methods work by identifying and rejecting redundant and meaningless vectors, which can be rejected without degrading the classification of their neighbors. The concept used in instance weighing and regression problems can be applied here in an analogous way as with noise filters.

1.3 Definitions

The two objectives of instance selection are minimization of the number of instances in the reduced training set **S** (we denote this set **S**, because it contains only **s**elected instances) and minimization of the error obtained on the test set by predictive models trained on the reduced training set **S**. The first objective is known as minimization of retention or maximization of reduction or compression. The second one in case of classification is expressed by the classification accuracy, that is the ratio of the correctly classified instances to all classified instances and in regression this is usually expressed with root mean square error ($rmse$) - the standard and commonly used measure of regressor performance. However, also other performance measures can be

used, as balanced accuracy in classification or the coefficient of determination R^2 in regression.

We use the following definitions in this book:

$$compression = 1 - \frac{N_{sel}}{N} = 1 - \frac{|S|}{|T|} \tag{1.3}$$

$$retention = \frac{N_{sel}}{N} = \frac{|S|}{|T|} \tag{1.4}$$

where N is the number of all instances in the training set $\mathbf{T}$, N_{sel} is the number of selected instances from $\mathbf{T}$, which create the selected set $\mathbf{S}$. Thus $retension = 1 - compression$.

We use the standard definitions of classification accuracy on the training set (acc_{trn}) and on the test set (acc_{tst}):

$$acc_{trn} = \frac{1}{N}\sum_{i=1}^{N}(\bar{C}(\mathbf{x}_i) = C(\mathbf{x}_i)) \text{ over the training set } \mathbf{T} \tag{1.5}$$

$$acc_{tst} = \frac{1}{N_{tst}}\sum_{i=1}^{N_{tst}}(\bar{C}(\mathbf{x}_i) = C(\mathbf{x}_i)) \text{ over the test set} \tag{1.6}$$

where $\bar{C}(\mathbf{x}_i)$ is the predicted and $C(\mathbf{x}_i)$ is the actual class of the i-th instance $\mathbf{x}_i$ (of the training set T in Eq. 1.5 and of the test set in Eq. 1.6), N is the number of all instances in the training set and N_{tst} in the test set.

We use the standard definitions of root mean square error on the training set ($rmse_{trn}$) and on the test set ($rmse_{tst}$):

$$rmse_{trn} = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\bar{Y}(\mathbf{x}_i) - Y(\mathbf{x}_i))^2} \text{ over the training set } \mathbf{T} \tag{1.7}$$

$$rmse_{tst} = \sqrt{\frac{1}{N_{tst}}\sum_{i=1}^{N_{tst}}(\bar{Y}(\mathbf{x}_i) - Y(\mathbf{x}_i))^2} \text{ over the test set} \tag{1.8}$$

where $\bar{Y}(\mathbf{x}_i)$ is the predicted and $Y(\mathbf{x}_i)$ is the actual value of the output variable of the i-th instance $\mathbf{x}_i$ (of the training set T in Eq. 1.7 and of the test set in Eq. 1.8), N is the number of all instances in the training set and N_{tst} in the test set.

1.4 Software Packages

We used several software packages while working on this book: the instance selection and neural network library, which we created in C# language (most operations), *RapidMiner*, including *Information Selection Extension*, *Weka*, *Mulan* (some operations) and some other software (very few operations). The main four packages, including the source code are available on-line by the links on the web page `kordos.com/inst-sel`. The webpage contains also the datasets and a lot of detailed experimental results, which are discussed in the book.

1.5 Datasets Used in this Book

To ensure reliable and unbiased results, we used the standard benchmark datasets from the KEEL Repository [8], as shown in Tables 1.4 and 1.3. The output values in the regression datasets were standardized in the experiments to enable us to evaluate and compare the results better. The tables present the properties of the datasets and the average $rmse$ or classification accuracy obtained in 10 runs of 10-fold cross-validation with 1-NN, k-NN with optimal k and MLP neural network. The architecture of the MLP neural networks and the learning algorithms will be presented in detail in the following chapters. Although also other learning models can be used, it is enough to show the performance of the three models to present the dataset properties, as the other models performance is approximately proportional on particular groups of datasets.

The properties of the multi-target datasets are listed in chapter 13, because those datasets are used only in that chapter. The last column in Table 1.4 contains the standard deviations of the $rmse$ obtained with optimal k-NN in 10-fold cross-validation. The standard deviations obtained with 1-NN and with MLP usually do not differ more than 15%. Also the standard deviations obtained on the compressed data rarely differ more than 15%. For that reason we will not report them in this book, but this information can be found in the additional resources at `kordos.com/inst-sel`. For the same reason we usually do not report the results of the statistical significance tests.

As it can bee seen from the tables there are some statistical dependencies between the accuracy or $rmse$ obtained on the test sets and the optimal k value in the k-NN algorithm. Higher optimal k corresponds to lower accuracy or higher $rmse$, this is to more noisy data. In noisy data the output value of many instances has more random nature than in high quality data. For that reason we need to average the outputs of more neighbors to make the random noise at least partially cancel-out, as we expect that the average value of the random component will tend to some small numbers as k increases. Using 1-NN algorithm, or even better 3-NN, based on the obtained $rmse$ we can predict the optimal k value. Moreover, we can also predict, which noise removal method will be optimal, as will be discussed in the third part of the book.

Table 1.3. Classification datasets used in the book (obtained from the Keel repository) and their properties: number of instances, number of attributes, the $accuracy$ obtained with 1-NN, the optimal k (the k in k-NN that gives the highest $accuracy$), $accuracy$ obtained with k-NN with optimal k and with MLP neural network in 10-fold cross-validation.

Dataset	**Inst.**	**Attr.**	**classes**	**acc(1-NN)**	**optK**	**acc(k-NN)**	**acc(MLP)**
iris	150	4	3	0.9395	3	0.9600	0.9733
sonar	208	60	2	0.8650	1	0.8650	0.7874
glass	214	9	6	0.8922	1	0.8922	0.9435
newthyroid	215	5	3	0.9861	1	0.9861	0.9630
spectfheart	267	44	2	0.7259	17	0.8194	0.7557
heart	270	13	2	0.8960	5	0.9145	0.8957
cleveland	297	13	5	0.8313	3	0.8316	0.8517
haberman	306	3	2	0.6589	15	0.7514	0.6957
bupa	345	6	2	0.7933	1	0.7933	0.8501
ionosphere	351	33	2	0.8743	1	0.8743	0.8946
movement_libr	360	90	15	0.8553	1	0.8553	0.8161
bands	365	19	2	0.7037	1	0.7037	0.6156
monk-2	432	6	2	0.9907	5	0.9930	1.0000
led7digit	500	7	10	0.6314	35	0.7535	0.7222
wdbc	569	30	2	0.9473	5	0.9701	0.9649
balance	625	4	3	0.7707	7	0.8797	0.9278
wisconsin	683	9	2	0.9575	9	0.9677	0.9559
pima	768	8	2	0.7042	25	0.7524	0.7394
mammograph	830	5	2	0.8263	7	0.8383	0.8636
vehicle	846	18	4	0.6960	5	0.7289	0.7945
yeast	1484	8	10	0.5334	17	0.5954	0.5967
titanic	2201	3	2	0.7391	11	0.7864	0.7814
img. segment.	2310	19	7	0.9662	1	0.9662	0.8104
spambase	4597	57	2	1.0000	1	1.0000	1.0000
banana	5300	2	2	0.8732	7	0.8953	0.7450
phoneme	5404	5	2	0.9063	1	0.9063	0.8466
page-blocks	5472	10	5	0.9788	1	0.9788	0.9740
texture	5500	40	11	0.9889	1	0.9889	0.9869
satimage	6435	36	6	0.9304	1	0.9304	0.9026
marketing	6876	13	9	0.5315	11	0.5356	0.5668
thyroid	7200	21	3	0.9579	1	0.9579	0.9890
ring	7400	20	2	0.7509	3	0.7512	0.9012
twonorm	7400	20	2	0.9753	1	0.9753	0.9953
coil2000	9822	85	2	0.8961	1	0.9546	0.7789
penbased	10992	16	10	0.9938	1	0.9938	0.9560
magic	19020	10	2	0.8192	15	0.8411	0.8607
letter	20000	16	26	0.9546	1	0.9546	0.9632
shuttle	57999	9	7	0.9992	1	0.9992	0.9991

Table 1.4. Regression datasets used in the book (obtained from the Keel repository) and their properties: number of instances, number of attributes, the $rmse$ obtained with 1-NN, the optimal k (the k in k-NN that gives the lowest $rmse$), $rmse$ obtained with k-NN with optimal k and with MLP neural network and the standard deviation of $rmse$ in 10-fold cross-validation with optimal k k-NN.

Dataset	**Inst.**	**Attr.**	**r0(1-NN)**	**optK**	**r0(k-NN)**	**r0(MLP)**	**std(k-NN)**
machineCPU	209	6	0.351	1	0.351	0.348	0.158
baseball	337	16	0.727	7	0.584	0.626	0.098
dee	365	6	0.555	7	0.424	0.407	0.052
autoMPG8	392	7	0.428	6	0.364	0.354	0.068
autoMPG6	392	5	0.407	4	0.366	0.357	0.066
ele-1	495	2	0.689	11	0.584	0.530	0.071
forestFires	517	12	1.547	11	0.864	0.735	0.651
stock	950	9	0.112	3	0.105	0.178	0.011
steel	960	12	0.348	4	0.323	0.225	0.086
laser	993	4	0.231	3	0.204	0.164	0.012
concrete	1030	8	0.533	4	0.521	0.379	0.041
treasury	1049	15	0.069	3	0.058	0.074	0.011
mortgage	1049	15	0.054	2	0.045	0.055	0.008
friedman	1200	5	0.462	7	0.340	0.318	0.021
wizmir	1461	9	0.230	7	0.178	0.085	0.021
wankara	1609	9	0.225	9	0.167	0.097	0.011
plastic	1650	2	0.617	30	0.468	0.435	0.016
quake	2178	3	1.344	50	1.025	1.000	0.040
anacalt	4052	7	0.227	2	0.212	0.201	0.075
abalone	4177	8	0.914	13	0.702	0.651	0.033
delta-ail	7128	5	0.716	17	0.560	0.552	0.043
puma32h	8191	32	1.212	21	0.895	0.338	0.022
compactiv	8192	21	0.254	2	0.231	0.152	0.048
delta-elv	9516	6	0.828	35	0.610	0.599	0.019
tic	9822	85	1.366	90	1.015	1.017	0.039
ailerons	13750	40	0.657	10	0.503	0.402	0.008
pole	14998	26	0.244	4	0.214	0.255	0.010
elevators	16598	18	0.686	8	0.559	0.344	0.023
california	20640	8	0.654	9	0.527	0.532	0.011
house	22784	16	0.872	11	0.687	0.710	0.030
mv	40767	10	0.210	9	0.140	0.055	0.002

Chapter 2
Instance Selection in Classification Tasks

Abstract We shortly present several instance selection algorithms for classification tasks. Then we discuss the problem of choosing the prediction model inside the instance selection algorithm. This model is in most cases k-NN but sometimes other models are preferred. We discuss also the computational complexity of instance selection algorithms and the possible ways of limiting it.

2.1 Introduction

Different families of instance selection methods use different approaches to that problem. We call the group of instance selection algorithms considered in this part of the book "similarity-based methods", because they use some similarity measures between the instances. The measures can be based on the distances calculated by the k nearest neighbor algorithm (k-NN) or the positions on Voronoi cells and they are used to assess which instances can be removed as noisy or redundant.

The two most popular instance selection algorithms are probably CNN and ENN. CNN (Condensed Nearest Neighbor) was the first instance selection algorithm developed by Hart in 1968. ENN (Eddited Nearest Neighgor) is a noise filter proposed in 1978 by Wilson.

These two algorithms were further extended leading to more complex ones like DROP1-5, IB3, GE, RNGE, ICF, ENRBF2, ELH, ELGrow, Explore and others, which are described in the next section. A large survey including almost 70 different algorithms of instance selection for classification tasks can be found in [9].

Usually the advantage of the similarity-based instance selection methods is the simplicity of implementation, smaller memory requirements and decoupling the selection process from a predictive model learning.

On the other hand the problem with classical instance selection algorithms is that in most cases they are based on certain assumptions and observations made by their authors about how the data is typically distributed. For example the ENN algorithm removes the instances misclassified by k-NN considering them noisy. That is not always true, as they may also be boundary instances required for correct classification and indeed ENN has the tendency to smooth the class boundaries. There are also other assumptions in other algorithms, which are more frequently true than not, but in some cases they are wrong what leads to sub-optimal results.

Typically a well designed instance selection process first removes noise to increase prediction accuracy. Then as more instances are removed to condense the data, the prediction accuracy begins to drop gradually, at the beginning very slowly and then faster, as shown in Fig. 2.1. However, only some of the instance selection methods allow us to adjust the degree of data reduction, in others we just get only one position on the plot.

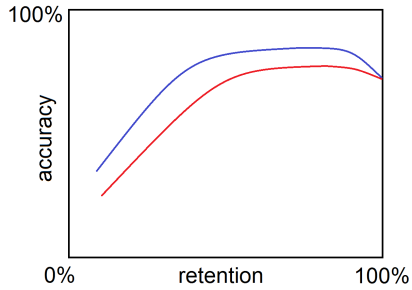


Fig. 2.1. Typical dependence between the percentage of instances remaining in the training set after instance selection (retention) and classification accuracy on the test set of the model trained on the reduced training set. The upper blue line – better instance selection algorithm, the lower red line – poorer instance selection algorithm.

2.2 Review of Selected Similarity-based Instance Selection Methods

Many similarity-based instance selection algorithms were proposed in the literature. In the short review we will present only the most popular and most effective ones.

Random selection. It is the simplest instance selection method, which randomly draws instances from the data set. The selection can be random or stratified to preserve

class label distribution [10].

CNN - Condensed Nearest Neighbor was proposed by Hart [11]. The purpose of CNN is to reject these instances, which do not bring any additional information into the classification process. CNN starts by adding a single randomly selected instance $\mathbf{x}_1$ from the original set **T** to the set of selected examples **S** and then tries to classify all other examples from **T** using k-NN (for the purpose of the classification **S** becomes now the training set and **T** the test set). If any instance from **T** is incorrectly classified it is moved to the selected set **T**, as it is believed to provide some additional information, which was missing in **S**, thus causing the misclassification of that instance. CNN is relatively fast and usually achieves an average level of compression. However, the quality of the solution depends on the level of noise in the data and on the random order, in which the instances are evaluated. For that reason for all condensation instance selection algorithms, including CNN it is advised to use a noise filter instance selection algorithm (e.g. ENN) to remove the noisy instances, before running them. A similar algorithm to CNN is CA. However, CA generates new instances by merging two neighborhood instances and placing the new instance in-between the original ones.

Algorithm 1 CNN algorithm

Require: $\mathbf{T}$
  $m \leftarrow |\mathbf{T}|$;
  $\mathbf{S} \leftarrow \mathbf{x}_1$;
  $p \leftarrow 0$;
  while $|\mathbf{S}| > p$ **do**
    $p \leftarrow |\mathbf{S}|$
    for $i = 1 \ldots m$ **do**
      $\bar{C}(\mathbf{x}_i) =$kNN$(\mathbf{S}, \mathbf{x}_i)$
      if $\bar{C}(\mathbf{x}_i) \neq C(\mathbf{x}_i)$ **then**
        $\mathbf{S} \leftarrow \mathbf{S} \cup \mathbf{x}_i$;
        $\mathbf{T} \leftarrow \mathbf{T} \setminus \mathbf{x}_i$;
      end if
    end for
  end while
  return $\mathbf{S}$

ENN - Editted Nearest Neighbor is a noise filter proposed by Wilson [12]. Its purpose is to increase classification accuracy by removing noisy instances from the training set, rather than to compress the data. ENN also uses k-NN to predict the class of each instance and marks the instances for which the predicted class is different than the real class, as these instances are considered noise. In the next step all marked instances

get removed from the training set. The compression obtained by ENN is usually very weak (typically at or below 10% of rejected instances). ENN usually works very well and is frequently used as the first step before running condensation instance selection algorithms. It must be clearly stated that the order of running these algorithms is very important and always the noise filter must be used first. This allows to remove the false nearest neighbors of the opposite class and false decision boundaries, which are the crucial concepts used by the condensation instance selection algorithms.

Algorithm 2 ENN algorithm

Require: $\mathbf{T}$
  $m \leftarrow |\mathbf{T}|$;
  for $i = 1 \ldots m$ **do**
    $marked_i = 0$;
    $\bar{C}(\mathbf{x}_i) =$kNN$(\mathbf{T} \setminus \mathbf{x}_i, \mathbf{x}_i)$;
    if $C(\mathbf{x}_i) \neq \bar{C}(\mathbf{x}_i)$ **then**
      $marked_i = 1$;
    end if
  end for
  for $i = 1 \ldots m$ **do**
    if $marked_i == 1$ **then**
      $\mathbf{T} = \mathbf{T} \setminus \mathbf{x}_i$;
    end if
  end for
  $\mathbf{S} \leftarrow \mathbf{T}$;
  return $\mathbf{S}$

GE - Gabriel Editing proximity graph based algorithm [13]. To determine if an instance $\mathbf{x}_b$ is a neighbor of instance $\mathbf{x}_a$ GE uses the following rule:

$$\bigvee_{a \neq b \neq c} D^2(\mathbf{x}_a, \mathbf{x}_b) > D^2(\mathbf{x}_a, \mathbf{x}_c) + D^2(\mathbf{x}_b, \mathbf{x}_c) \tag{2.1}$$

where $D(\mathbf{x}_a, \mathbf{x}_b)$ is the distance in the graph between the two instances $\mathbf{x}_a$ and $\mathbf{x}_b$. If the two instances are neighbors and the class of $\mathbf{x}_a$ and all its neighbors is equal, then $\mathbf{x}_a$ is marked for removal. As CNN, also GE is sensitive to noise and outliers. However, GE usually allows to obtain stronger compression.

RNG - Relative Neighbor Graph algorithm [13] works in a similar way as GE, however it uses another cost function and the function used to evaluate the neighbors is modified as follows:

$$\bigvee_{a \neq b \neq c} D(\mathbf{x}_a, \mathbf{x}_b) \geq \max(D(\mathbf{x}_a, \mathbf{x}_c), D(\mathbf{x}_b, \mathbf{x}_c)) \tag{2.2}$$

All-kNN - All-kNN is another version of a repeated ENN algorithm, which repeats the ENN procedure, each time with a different k value, thus rejecting more instances [14]. Depending on a given problem its results may be better or worse than those of the base ENN algorithm.

RENN - Repeated ENN repeats ENN so long until no instance is removed [14]. It has similar noise reduction properties as the ENN algorithm and depending on a given problem its results may be better or worse than results of the base ENN algorithm.

Algorithm 3 RENN algorithm

Require: $\mathbf{T}$
  $flag \leftarrow$ **true**
  while flag **do**
    $flag \leftarrow$ **false**
    $\bar{\mathbf{S}} =$ENN$(\mathbf{S}, \mathbf{T})$
    if $\bar{\mathbf{S}} \neq \mathbf{S}$ **then**
      $flag \leftarrow$ **true**
    end if
  end while
  return $\mathbf{S}$

IB2 and IB3 - Instance Based Learning was proposed by Aha [15]. The IB2 is a simplified one pass CNN algorithm. The IB3 additionally evaluates instances according to the following formula for the upper and lower bound:

$$Bound = \frac{p + z^2/2n \pm z\sqrt{p(1-p)/n + z^2/4n^2}}{1 + z^2/n} \tag{2.3}$$

where n is the number of attempts, p is the accuracy of the attempts, and z is the confidence level for the instance and p is the frequency, n is the number of processed instances, and z the confidence level for the frequency of a class. An instance is selected if the lower bound with the confidence level 0.9 is greater than the upper bound of the frequency of its class label and the instance is rejected if the upper bound of its accuracy is lower with the confidence level 0.7 than the lower bound of the frequency of its class. IB3 usually achieves strong compression and is insensitive to noise and outliers. According to our tests, the results obtained with ENN followed by IB3 were comparable to that of DROP3 and DROP5, while the execution time was about three times shorter (Table 2.1).

ELH - Encoding Length Heuristic was proposed by Cameron Jones [16]. It evaluates the influence of an instance rejection on classification by applying the following formula:

$$J(m,n,z) = F(m,n) + m\log_2(c) + F(x,n-m) + x\log_2(c-1) \tag{2.4}$$

where n and m are numbers of instances in the training set and in the new selected dataset **S** respectively, x is the number of wrongly classified instances (using **S** as the training set) and $F(m,n)$ is a cost of coding n examples by m examples.

$$F(m,n) = \log^* \left(\sum_{j=0}^{l} \frac{n!}{j!(n-j)!} \right) \tag{2.5}$$

where $\log^*$ is a cumulative sum of positive factors $\log_2$. ELH first rejects an instance and then evaluates the influence of the rejection on the ELH heuristic. If the rejection does not decrease the value of the heuristic the instance gets finally rejected and the procedure is repeated with the next instance. ELH achieves strong compression with usually high accuracy and is insensitive to noise and outliers. Another versions of ELH are ELGrow and Explore and an algorithm called DEL is a decremental version of ELH.

MC - Monte-Carlo selection is an extension of random selection [10]. MC repeats the random selection given number of times and selects the best of the subsets.

RMHC - Random Mutation Hill Climbing is another algorithm based on random selection [17]. RMHC uses two parameters - the number of selected instances and the number of iterations.

RMHC encodes the instances into a binary array. To encode a single instance $\lceil \log_2(n) \rceil$ bits are needed, where n is the number of instances in the training set. Thus $c \cdot \lceil \log_2(n) \rceil$ bits are required to encode c instances. The algorithm in each iteration randomly mutates a single bit and checks the accuracy obtained by k-NN using the current training set. If the mutation improves the accuracy, then the new solution is kept, if not - RMHC returns to the previous solution. This is repeated the defined number times. Also the number of selected instances is defined so in this way, we can directly influence the compression level. RMHC is also insensitive to noise and outliers.

DROP. The DROP methods (DROP1 to DROP5) [12] belong to the instance selection algorithms that produce the best results in classification tasks [18, 19]. The algorithms belong to decremental methods and the rejection decision is taken considering the nearest neighbors and associates of the given instance. The nearest neigh-

Algorithm 4 RMHC algorithm

Require: $\mathbf{T}$, $numSelected$, $numIter$
  $m \leftarrow |\mathbf{T}|$
  $st \leftarrow Bin(l \log_2 m)$
  $st^* \leftarrow SetBit(st, numSelected)$
  for $i = 1 \ldots maxit$ **do**
    $\mathbf{S}^* \leftarrow GetProto(\mathbf{T}, st^*)$
    $tacc \leftarrow Acc(1NN(\mathbf{S}^*, \mathbf{T}))$
    if $tacc > acc$ **then**
      $acc \leftarrow tacc$
      $\mathbf{S} \leftarrow \mathbf{S}^*$
      $st \leftarrow st^*$
    end if
    $st^* \leftarrow MutateBit(st)$
  end for
  return $\mathbf{S}$

bors of an instance x are the k instances closest to x. The associates of an instance x are those instances that have the instance x as one of their k nearest neighbors.

The DROP algorithms work in the following way:

- DROP1 eliminates an instance x, if this does not affect the classification of its associates.
- DROP2 before starting the selection sorts the instances in descending order by the distance from the instance to its nearest enemy (an instance from another class). So first the instances located among the same class instances are processed and later the instances close to the class boundary.
- DROP3 additionally first applies a noise filter that works like the ENN algorithm, removing the instances incorrectly classified by k-NN.
- DROP4 applies a modified version of the noise filter, by additionally verifying if removing an instance does not cause a misclassification of another instance and if it does, the instance will not be removed.
- DROP5 is similar to DROP3, but instead of using the noise filter, it starts the analysis from the instances that are closest to the nearest enemies (from those close to the class boundaries). In this way as a matter of fact the noise filter is applied at the first stage. However, its computational complexity is still $O(n^3)$.

LVQ - Learning Vectors Quantization is a model proposed by Kohonen in [20]. In contrary to all previous algorithms (except CA) LVQ changes the positions of the prototype vectors during learning and finally they have different values than original instances of the training set.

ICF - Iterative Case Filtering [21] is a two-step algorithm. First it applies the ENN algorithm to remove noisy instances. Then it identifies the local sets. A local set of an instance $\mathbf{x}_i$ is a set of all instances which are closer to $\mathbf{x}_i$ than the nearest neighbor of $\mathbf{x}_i$ from a different class, also called nearest enemy. ICF uses local sets to create two sets of instances: coverage and reachability, in a similar way, as the nearest neighbor and associates used in DROP algorithms. The coverage of an instance is its local set, which includes all instances that are closer to the instance than its closest enemy. The reachability of an instance $\mathbf{x}_i$ is the set of associates, this is these instances, for which $\mathbf{x}_i$ is one of their nearest neighbors. If $Reachability(\mathbf{x}_i) > Coverage(\mathbf{x}_i)$ then the instance $\mathbf{x}_i$ is removed from $\mathbf{T}$. In this way ICF removes an instance if the information it carries can be expressed by other instances located around it.

LSBo - Local Set Border Selector presented by Leyva et. al. [22] is also based on local sets, and it displays some similarities to ICF. This is also a hybrid approach, which uses a heuristic criterion: the instances close to the boundaries between classes tend to have greater local sets. LSBo starts from a noise filtering algorithm called LSSm, which was also presented in [22].

HMNE - Hit Miss Network Editing [23] first constructs a Hit Miss network (HMN), this is a graph, in which each instance $\mathbf{x}_i$ is a vertex V_i and each edge connects to the nearest neighbor of each class. Thus the degree of each vertex equals the number of classes. Each vertex is represented by two counters *hit* and *miss*. The hit counter $Hit(V_i)$ of the nearest neighbor $\mathbf{x}_j$ is increased if the two instances belong to the same class and the miss counter $Miss(V_i)$ if they belong to different classes. Then an instance is marked for removal when in-degree$(V_i) = 0$, or when $|Miss(V_i)| \geq |Hit(V_i)|$. If too many instances were marked for removal from one class, an un-mark process of all instances of that class is carried out. If the number of classes $c > 3$ and $|Miss(V_i)| < c/2$ then all instances that satisfy in-degree$(V_i) > 0$ are un-marked. If $Hit(V_i) > 25\%$ of instances belonging to the class y_i also un-mark is performed. In the last step the algorithm removes all marked instances.

CCIS - Class Conditional Instance Selection [23] consists of two steps. In the first step the HMN graph is constructed and used by the K-divergences instance scoring function. The instances with negative or zero score are rejected, then instances with the highest scores are iteratively selected until the error of the k-NN starts increasing. In the second step the co called thin-out selection is performed, where only instances close to the decision boundary are selected. This step further improves compression.

ATISA - Adaptive Threshold-based Instance Selection Algorithm [24]. ATISA1 first applies a noise filter to the dataset and then calculates thresholds for the instances. Each remaining instance is classified with 1-NN. In the case of an incorrect classification, the instance is added to the final set $\mathbf{S}$. In case of a correct classification, it is

added only if the distance to its nearest neighbor is greater than the threshold of that neighbor.

The second version of the algorithm (ATISA2) instead of randomly selecting the instances (as ATISA1 does), considers the distances in decreasing order of the distance between the instance and its nearest enemy.

ATISA3 calculates the threshold dynamically at each instance selection. This causes higher reduction rates, as the dynamic threshold update increases the chances of an instance to be within the coverage area of another instance.

CNNIR - Constraint Nearest Neighbor-Based Instance Reduction [25]. CNNIR first eliminates noise by removing instances without natural neighbors and then searches for core instances using a constraint nearest neighbor chain. Core instances are the most important instances inside the class. The chain consists of three instances: the root pattern, which is a randomly selected instance x_i from the dataset, the second pattern, which is the nearest enemy of x_i and the nearest neighbor from the same class. The chain is used to select border instances which can construct a rough decision boundary. In the next step a specific strategy is used to reduce the border set. In the last step the selected instances are obtained by merging border and core instances.

Instance Rebelling - Instead of directly selecting instances from the training data an interesting approach for training k-NN classifier was proposed by Kuncheva in [10], where preselected instances were relabelled, such that each instance was assigned to all class labels with appropriate weights describing the support for given label.

Anomaly Detection - The anomaly detection algorithms were originally designed for unlabeled data. There are a lot of anomaly detection methods and a survey of them can be found in [26] and the most popular methods based on nearest neighbors include: k-NN Global Anomaly Score, Local Outlier Factor (LOF), Connectivity based Outlier Factor (COF), Local Outlier Probability (LoOP), Influenced Outlierness (INFLO), Local Correlation Integral (LOCI) [27].

k-NN Global Anomaly Score is probably the most widely used of theses methods. The anomaly score is either calculated as the average distance $d_{x,n}$ of the k nearest neighbors or to the distance to the k-th neighbor. The first method is more robust to noise in data and thus we use this version.

In case of labeled data we need both distances: in the input space $d_{x,n}$ and in the output space $d_{y,n}$. The purpose is also the same: to make more outstanding instances less influence the model training and we also need to adjust the coefficient to local data density α, which will be discussed in the next chapter. We use Euclidean distance measure to calculate the distances $d_{y,n}$ and $d_{x,n}$ and as the number of neighbors we empirically propose to use $k = 9$. Thus the modified global anomaly score of the n-th instance can be defined in two ways:

$$w_{out,n} = d_{y,n}/d_{x,n} \tag{2.6}$$

The second possibility to adjust the method to labeled data is to treat the output value as one of the inputs, however with a different, usually higher user-defined weight and use the standard definition for unlabeled data.

2.3 Reducing Computational Complexity

The computational complexity of most similarity-based instance selection methods is between $O(n^2)$ and $O(n^3)$.That is because finding the nearest neighbors or the vertices of a graph requires usually $O(n^2)$ operations. For large datasets that can be a real problem.

One solution to this problem is to partition the dataset into several smaller parts and perform instance selection independently in each part. The downside of this approach is the possible accuracy loss at the boundaries of the partitions, where some of the nearest neighbors of some instances may fall into another partition and thus not be taken into account in the calculations. In regression it may not be a big problem, but in classification it is and therefore some methods to deal with it were proposed [22, 28].

The locality-sensitive hashing (LSH) is a method for determining similarity between elements. It makes use of hash functions, which try to assign similar items to the same bucket with a high probability, and at the same time to decrease the probability of assigning dissimilar items to the same bucket [18]. LSH is frequently used to improve the efficiency of nearest neighbors calculation [29]. Thus the first benefit of LSH for instance selection algorithms is the speeding up of nearest neighbor calculation, which is performed by most of similarity-based instance selection algorithms. However, that does not change the complexity of the instance algorithm itself. In [30] the use of LSH was proposed, not only for the calculation of nearest neighbors, but also for the core of the instance selection algorithm. The idea was to perform instance selection within each bucket obtained with LSH. This allowed the instance selection to obtain linear complexity.

Similar solutions based on clustering will be discussed in the second part of this book in the context of evolutionary instance selection for regression problems, where it can not only accelerate the calculations but also improve the results.

2.4 Other Evaluation Models

Many of the instance selection algorithms use k-NN or even originally they used 1-NN as the evaluation model. However, for obtaining the best results, the evaluation model should be the same model as the final classifier. For example if the final classifier is

5-NN, the evaluator within the instance selection algorithm should be also 5-NN, if some kind of weighted k-NN is used as the final classifier, the same weighed k-NN should be used as the inner evaluator, if the final model is an MLP neural network, the inner evaluator should be also an MLP network. That is illustrated in Fig. 2.2, where k-NN cannot perform proper classification in some conditions, yet for other methods, as e.g. for decision trees this is not a problem. The downside of using other inner evaluators than k-NN is that learning other models as many time as the number of instances in the training set is very time consuming. However, this approach after some improvements can be used in regression tasks, as will be presented in the next chapter.

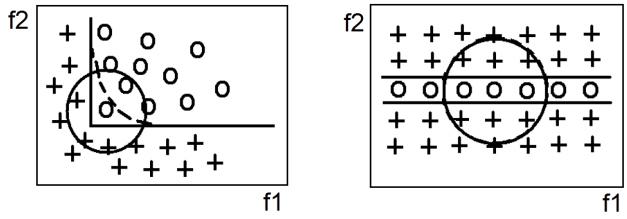


Fig. 2.2. Examples of decision borders that make some instances to be incorrectly classified by k-NN and which influence also the instance selection, where k-NN is the inner evaluator. Cross and circle denote instances of two different classes, with two features $f1$ and $f2$ corresponding to the positions in horizontal and vertical directions.

2.5 Comparison of Selected Similarity-based Instance Selection Methods

Most research in instance selection algorithms done so far was in the area of similarity-based instance selection algorithms for classification tasks. For that reason we focused our research on other aspects of instance selection, as instance weighting schemes, ensembles, regression problems, evolutionary, and embedded methods.

Our contribution in developing the classical instance selection algorithms, which can be applied for classification tasks included adjusting k-NN Global Anomaly Score algorithm to labeled data, instance weighting schemes and adjustments of inner evaluation models. However, we mostly focused on application of these techniques in regression problems and therefore they are presented in the following chapters. In Table 2.1 we present some experimental results, taking into account also the computational time of particular algorithms.

Table 2.1. Average results over the 10 datasets (ionosphere, image segmentation, magic, thyroid, page-blocks, shuttle, sonar, satellite image, penbased, ring) classification accuracy of MLP neural network trained on the set of selected instances, percentage of selected instances and execution time relative to CNN time of instance selection process for most popular instance selection algorithms using the RapidMiner implementation [31, 32]. * stands for user defined compression

Method	accuracy	%Instances	time	complexity	type
no selection	92.74	100	-	-	-
GE	90.71	45	130	$O(n^3)$	decremental
RNG	86.81	12	10	$O(n^3)$	decremental
CNN	86.74	8.0	1.0	$O(n^3)$	incremental
IB3	86.56	4.0	3.5	-	-
DROP-3	87.13	4.0	14	$O(n^3)$	decremental
MC	82-86	3.5-20*	8.0	$O(n^2)$	mixed
RHMC	82-86	3.5-20*	8.1	$O(n^2)$	mixed
ENN+CNN	87.44	7.1	2.0	-	-
IB2	85.12	7.7	0.2	$O(n^2)$	incremental
IB3	-	-	-	$O(n^2 log_2 n)$	incremental
ENN+IB3	87.15	3.9	4.5	-	-
ENN	93.17	90	1.0	-	-

Several studies and comparisons of the classical instance selection algorithms for classification problems can be easily found in literature [19, 33, 3]. Thus there is no need to repeat these experiments here once again. The general conclusion from those studies is that there is no single instance selection algorithm, which performs best for most datasets and for most classifiers. Nevertheless, a group of algorithms that usually outperform the remaining ones can be determined. In [19] several instance selection algorithms were evaluated with several predictive models (k-NN, NRBF, FSM, IncNet, SSV, SVM). The authors concluded that "Explore, RMHC, MC, LVQ, DROP2-4 and DEL are the most effective instance selection algorithms. They automatically estimate the number of instances for optimal compression of the training set and reach high accuracy on the unseen data. (...) In the group of noise filters ENN algorithm came at the top." It is also worth pointing out that $O(n^3)$ is the most pessimistic assessment of the complexity of CNN, while in practice it is in most cases much closer to $O(n^2)$. Taking into account that one operation of CNN is relatively fast, it makes CNN in practice one of the fastest instance selection algorithms.

2.6 Conclusions

Similarity-based instance selection algorithms for classification tasks are historically the first group of developed methods. The best known and most widely used instance selection methods belong to this group. Also most of the research in instance selection has been focused so far on this group, including several big comparison studies available in the literature. We started the book from presenting the representatives of this group first for the complementarity of this work and second, because in further chapters we describe some new approaches, which are developed based on these methods. Also we present further some comparisons between similarity-based instance selection algorithms for classification and instance selection algorithms from the other groups.

Chapter 3
Instance Selection in Regression Tasks

Abstract We discuss methods of instance selection for regression tasks. The first method uses discretization of the output variable and converts the task to a multi-class classification. Then an instance selection method for classification is used and finally the data is converted back to regression problem. The next solution uses a distance threshold between instances. If two instances are further from each other than the threshold then they are considered by the instance selection algorithm in the same way as different class instances. We also shortly review other approaches.

3.1 Introduction

Instance selection for regression tasks is a more complex problem than for classification tasks. For a classifier to work correctly only the boundaries between classes must be precisely determined. In regression tasks there are no classes and thus no class boundaries, and output value must be properly calculated at each point of the data space. Thus all the data space is important and not only some selected parts as in classification. For that reason the dataset compression obtained in classification problems can be much stronger than in non-linear regression. The next point is that the prediction result in classification tasks is either binary or there are at most several different classes, while in regression tasks, the prediction result is continuous, which correspond to an unlimited number of possible values.

Thus, the standard instance selection approaches designed for classification require some modifications to make them work with regression problems. We discuss in this chapter two such modifications: setting a similarity threshold, which replaces the "same class" concept and performing discretization of the output variable to convert the problem to a multi-class classification. In the last section we will shortly review some other possible options presented in literature

We also discuss the problem that in a similar way as in classification tasks, for obtaining the best results, the evaluation model should be the same model as the final classifier.

3.2 Threshold-based Instance Selection for Regression Tasks

Since in regression there are no classes, the concept of a class can be replaced by some threshold distance [34, 35, 30] or probability density function [36]. If the difference between the real output value $Y(\mathbf{x}_i)$ of instance $\mathbf{x}_i$ and the predicted output value $\bar{Y}(\mathbf{x}_i)$ is greater than the threshold θ - the instances can be treated by the instance selection algorithm the same way as different class instances in classification tasks. If the distance is smaller than the threshold then as the same class instances.

$$\text{if } (Error = |Y(\mathbf{x}_i) - \bar{Y}(\mathbf{x}_i)| > \theta) \text{ then } (\dots). \tag{3.1}$$

where $(\dots)$ is the consequence of the rule, and it can denote either instance acceptance or rejection, depending on a given algorithm.

Now the problem is to adjust the threshold θ properly. Two factors should be considered here: the purpose of instance selection - noise reduction or data condensation and if for a given purpose the threshold should be constant or variable dependent on the point in the data space.

Per analogy to classification tasks, if the purpose of selection is to remove noise then we want to remove the instances, which differ too much from their neighbors and when the purpose is to reduce the data size, we want to reject the instances that are too similar to their neighbors, as shown in Fig. 1.1.

However, there are areas of naturally lower and higher diversity in the data space. In lower diversity areas, even a slight deviation from the value predicted by k-NN can mean that an instance is an outlier and should be rejected. While in higher diversity areas, such deviations can be normal. The experiments confirmed that to obtain the best results as well for noise removal as for data reduction, the threshold should not be constant, but proportional to the local diversity of the data [34]. For that purpose, we made the threshold θ proportional to the standard deviation of the output values of k nearest neighbors of the instance - $std(neighborhood(\mathbf{x}_i))$. When $std(neighborhood(\mathbf{x}_i))$ is high, the threshold θ should be less sensitive to the values of errors as shown in Eq. (3.1). In other cases, when $std(neighborhood(\mathbf{x}_i))$ is small, the threshold θ should also be smaller.

$$\theta = \alpha * std(neighborhood(\mathbf{x}_i)) \tag{3.2}$$

For the regression version of the ENN algorithm [37], the threshold θ should be relative large, as this algorithm is a noise filter. For the compression purpose a point

very similar to the neighbors can be rejected as it does not bring any useful information, so for the regression version of the condensation instance selection algorithms, the threshold θ should be small.

Based on the experiments, we can recommend α = 5.0 for noise reduction and α = 0.2 for data condensation as a good starting point, which can be used unless there are some special requirements. By adjusting the α parameter we can make the instance selection stronger or weaker. This is an additional advantage of this approach.

The pseudo-code of threshold-based regression ENN (T-ENN) is shown in Algorithm 5. T-ENN iterates over the instances of the training set and in each iteration one instance x_i is examined, so it is temporally removed from the training set and added to the one element test set (leave-one-out). This iteration is the same as in the standard ENN for classification tasks. $Model$ is the method used for prediction (any regression algorithm can be used), $Y(\mathbf{x}_i)$ is the actual target value of instance $\mathbf{x}_i$, $Model\left((\mathbf{T}, neighborhood(\mathbf{x}_i)) \setminus \mathbf{x}_i, \mathbf{x}_i\right)$ is the predicted value of the target output given by the model trained on dataset $\mathbf{T}$ without $\mathbf{x}_i$ and $neighborhood(\mathbf{x}_i)$ contains k nearest neighbors of the instance $\mathbf{x}_i$. The threshold θ is a product of $std(neighborhood(\mathbf{x}_i))$ and α. The parameter α is given as input to the algorithm. The remaining training set ($\mathbf{T} \backslash x_i$) is used to predict the output value $Y(x_i)$ of the instance x_i with the model. If the error (the difference between the predicted value $\bar{Y}(\mathbf{x}_i)$ and the actual value $Y(\mathbf{x}_i)$ is greater than the threshold θ, the instance x_i is marked for rejection. In the next iteration the instance x_i is returned to the original dataset $\mathbf{T}$ and the procedure is subsequently repeated with all the other instances. The selected dataset $\mathbf{S}$ consists of the instances, which were not marked for rejection.

The pseudo-code of threshold-based regression CNN (T-CNN) is presented in Algorithm 6. The purpose of T-CNN is to compress the dataset. T-CNN uses a threshold θ defined in the same way as in T-ENN , but the α parameter is much lower, typically α=0.2. As CNN for classification, T-CNN starts from one randomly selected instance. In each iteration when difference between the predicted output value $\bar{Y}(\mathbf{x}_i)$ and the real value $Y(\mathbf{x}_i)$ is greater than θ the instance is added to the selected set $\mathbf{S}$. Generally, lower α (and consequently lower θ) in T-CNN leads to the acceptance of more instances.

This approach can also be introduced into other instance selection algorithms. Based on our proposal, in [38] the authors adapted DROP2 and DROP3 to regression tasks using the same concept of threshold as we presented. They also considered the error accumulation approach (comparing the accumulated error that occurs when an instance is selected and when it is rejected), but the threshold based approach worked better.

Algorithm 5 T-ENN algorithm

Require: $\mathbf{T}, \alpha$
  $m \leftarrow |\mathbf{T}|$;
  for $i = 1 \ldots m$ **do**
    $marked_i = 0$;
    $\bar{Y}(\mathbf{x}_i) = Model\left((\mathbf{T}, \mathbf{neighborhood}(\mathbf{x_i})) \setminus \mathbf{x}_i, \mathbf{x}_i\right)$;
    $\theta = \alpha \cdot std(neighborhood(\mathbf{x}_i))$
    if $|Y(\mathbf{x}_i) - \bar{Y}(\mathbf{x}_i)| > \theta$ **then**
      $marked_i = 1$;
    end if
  end for
  for $i = 1 \ldots m$ **do**
    if $marked_i == 1$ **then**
      $\mathbf{T} = \mathbf{T} \setminus \mathbf{x}_i$;
    end if
  end for
  $\mathbf{S} \leftarrow \mathbf{T}$
  return $\mathbf{S}$

Algorithm 6 T-CNN algorithm

Require: $\mathbf{T}, \alpha$
  $m \leftarrow |\mathbf{T}|$;
  $\mathbf{S} \leftarrow \mathbf{x}_1$;
  $p \leftarrow 0$;
  while $|\mathbf{S}| > p$ **do**
    $p \leftarrow |\mathbf{P}|$
    for $i = 1 \ldots m$ **do**
      $\bar{Y}(\mathbf{x}_i) =$Model$((\mathbf{S}, \mathbf{neighborhood}(\mathbf{x_i}))\mathbf{x}_i)$;
      $\theta = \alpha \cdot std(neighborhood(\mathbf{x}_i))$
      if $|Y(\mathbf{x}_i) - \bar{Y}(\mathbf{x}_i)| > \theta$ **then**
        $\mathbf{S} \leftarrow \mathbf{S} \cup \mathbf{x}_i$;
        $\mathbf{T} \leftarrow \mathbf{T} \setminus \mathbf{x}_i$;
      end if
    end for
  end while
  return $\mathbf{S}$

3.3 Discretization-based Instance Selection for Regression Tasks

Another option to adjust instance selection algorithms for classification tasks to regression problems is to perform discretization and convert the regression problem to a multi-class classification task [39, 38]. This allows to directly use the standard instance selection algorithms for classification tasks. The first step in the process is the output variable discretization. In this way the problem is transposed into a classification task. Then the instance selection is performed using the instance selection algorithms for classification tasks and finally the numerical values of the output variable of the selected instances are restored. This methodology, in a similar way as the threshold based approach, can be considered a kind of meta-algorithm as it allows to use many instance selection methods for classification, which were presented in the previous chapter.

The whole process consists of the following steps:

1. Discretize the output value of instances in the dataset **T**.
2. Use an instance selection algorithm for classification to the discretized dataset to obtain the selected set **S**.
3. Restore the original numerical output values of instances in the selected set **S**.

Proper discretization is a very important point in this method, because it determines the boundaries between classes and the boundaries determine particular instance selection or rejection. There are two main categories of discretization algorithms: supervised (the class value is taken into account) and unsupervised or class-blind (the class value is not considered). In this case only the unsupervised algorithms can be used, because in the original data there are no classes and the discretization refers to the output variable. Two frequently used unsupervised discretization algorithms are equal-frequency and equal-width methods. We used the equal-width method either with a constant number of 10 bins or with different number of bins for noise reduction and data compression or with the optimized number of bins using the method of the estimated entropy, where different number of bins up to a maximum number b are tested and the number, which results with the lowest entropy is finally selected.

3.4 Data Partitioning

As it was already mentioned, in regression problems the value of each point in the data space needs to be determined, unlike in classification, where only the points determining the boundaries between classes matters. For that reason it is possible to split the dataset into several clusters and predict the output of an instance x by learning the model only on this cluster to which the instance belongs. There may be still a problem if the instance is very close to the cluster boundaries, so that some of its neighbors

belong to a different cluster and the model is not learned on that data. Fig. 11.4 shows the data that has to be evaluated by k-NN to determine the selection or rejection of all instances. This figure assumes that all the clusters are of equal size, what does not have to be always satisfied, that however does not change the idea. After performing the clustering, which has a linear computational complexity, we have to calculate only several small distance matrices (marked in orange) for k-NN (which have quadratic complexity), instead of one big matrix. To include in the training set the boundary instances from the adjacent clusters (marked in yellow in Fig. 11.4), we have to calculated some more distances, but still much fewer that without the partitioning. If we use a different model than k-NN to evaluate the instance, we also have to train this model each time only on the instances marked either only in orange or in both orange and yellow.

The idea of data partitioning is discussed in more detail in chapter 10 in the second part of the book, while discussing evolutionary instance selection, as in that case it has even more benefits.

3.5 Experimental Evaluation

The purpose of the presented experiments it to verify the following statements:

1. The best results can be obtained when the same evaluation model is used inside the instance selection process and as the final predictor.
2. An MLP neural network used as the inner evaluator inside the instance selection process without loss of accuracy can be trained only on a limited number of instances - those instances that are the closest neighbors of the point of interest.
3. Variable Θ parameter improves the results in comparison to constant Θ.

In this section we present only the results obtained with threshold-based instance selection for regression tasks. The results obtained with discretization-based approach will be presented and compared with the threshold-based approach in the next chapter.

In regression tasks, as in classification tasks, the best results can be obtained if the inner evaluation model is the same as the final predictor model. However, in regression the computational cost of using other models than k-NN does not have to be so high, because the models can be trained only on the part of the dataset that is close to the current instance of interest. That is because the instances that are far from the current instance have little or no impact on its predicted value.

This allowed us to successfully use an MLP neural network as the inner evaluation model of the T-ENN and T-CNN algorithms, where the final predictor was also an MLP neural network.

We used the following inner evaluation models to predict the output value of the instances inside the instance selection algorithm:

- k-NN with optimal k for each dataset (Ek, C, ECk in Tables 3.1 and 3.2)
- MLP network trained on the entire available training data within one validation of the cross-validation process (EM1, CM1, ECM1 in Tables 3.1 and 3.2)
- MLP network trained on about 30 percent of the training instances, which were closest to the considered instance (EM.3, CM.3, ECM.3 in Tables 3.1 and 3.2)
- MLP network trained on about 10 percent of the training instances, which were closest to the considered instance (EM.1, CM.1, ECM.1 in Tables 3.1 and 3.2)

However, since T-CNN is an incremental algorithm, there were too few instances at the begging of the optimization to train an MLP network (the process starts from a single instance), so k-NN (with k=1, as there was a single instance selected at the very beginning) was used as long as there were fewer than 10 instances selected.

Both MLP networks in the experiments (the network used for instance selection and the network used for the final prediction) had one hidden layer with sigmoid activation function and the number of hidden neurons was rounded to 50% of the number of attributes, but no less than 3. Both networks (the inner evaluator and the final predictor) were trained with the Rprop algorithm for 150 training cycles with standard Rprop parameters.

Table 3.1. Experimental results: relative $rmse$ (as % of $rmse$ without instance selection) for instance selection in regression tasks with T-ENN, T-CNN and T-ENN + T-CNN: with inner evaluation algorithm optimal-k k-NN and MLP trained on 100%, 30% and 10% on the training dataset. Prediction algorithm: MLP neural network. Ek: T-ENN with inner k-NN, EM1: T-ENN with inner MLP trained on 100% instances, EM.3: T-ENN with inner MLP trained on 30% instances, EM.1: T-ENN with inner MLP trained on 10% instances. Ck, CM1, CM.3, CM.1: the same for T-CNN, ECk, ECM1, ECM.3, ECM.1: the same for T-ENN followed by T-CNN.

dataset	Ek	EM1	EM.3	EM.1	Ck	CM1	CM.3	CM.1	ECk	ECM1	ECM.3	ECM.1
autoMPG8	0.989	0.957	0.963	0.987	1.172	1.094	1.073	1.187	1.116	1.095	1.102	1.130
autoMPG6	0.964	0.938	0.950	0.965	1.045	0.963	0.948	1.017	0.970	0.962	0.985	0.963
ele-1	0.942	0.932	0.944	0.956	1.045	1.016	1.029	1.014	0.946	0.930	0.945	0.956
stock	0.992	0.986	0.962	0.994	1.069	0.968	0.950	0.994	1.010	1.001	1.002	0.993
wankara	1.013	1.021	1.025	1.045	1.078	0.990	1.001	1.020	1.048	1.042	1.071	1.067
plastic	1.016	1.011	1.027	1.006	1.045	0.951	0.947	0.969	1.008	1.022	0.997	1.015
quake	1.020	1.021	1.042	0.994	1.141	1.086	1.094	1.111	1.104	1.092	1.123	1.117
anacalt	0.931	0.909	0.921	0.881	1.045	0.934	0.964	0.924	0.926	0.908	0.908	0.921
delta-ail	1.024	1.037	1.036	1.064	1.108	1.029	1.015	1.019	1.082	1.091	1.105	1.112
elevators	1.052	1.045	1.032	1.056	1.160	1.079	1.101	1.057	1.169	1.152	1.131	1.163
california	1.019	0.983	0.974	0.961	1.092	1.022	1.000	1.014	1.059	1.036	1.025	1.061
house	1.046	1.034	1.054	1.017	1.174	1.120	1.145	1.090	1.173	1.182	1.169	1.178
average	1.001	0.989	0.994	0.994	1.098	1.021	1.022	1.035	1.051	1.043	1.047	1.056

Table 3.2. Experimental results: retention for the same experiments as in table 3.1.

dataset	Ek	EM1	EM.3	EM.1	Ck	CM1	CM.3	CM.1	ECk	ECM1	ECM.3	ECM.1
autoMPG8	0.882	0.837	0.828	0.844	0.783	0.723	0.738	0.757	0.695	0.628	0.618	0.605
autoMPG6	0.920	0.875	0.865	0.845	0.904	0.837	0.844	0.813	0.755	0.725	0.705	0.753
ele-1	0.836	0.761	0.735	0.780	0.862	0.762	0.778	0.784	0.700	0.691	0.712	0.718
stock	0.903	0.803	0.787	0.811	0.783	0.721	0.725	0.787	0.737	0.711	0.697	0.705
wankara	0.659	0.636	0.625	0.659	0.776	0.770	0.760	0.755	0.591	0.550	0.541	0.558
plastic	0.820	0.766	0.781	0.752	0.817	0.784	0.808	0.808	0.664	0.600	0.574	0.601
quake	0.810	0.780	0.800	0.793	0.783	0.756	0.772	0.759	0.737	0.707	0.718	0.714
anacalt	0.690	0.650	0.631	0.641	0.909	0.843	0.871	0.813	0.586	0.588	0.597	0.560
delta-ail	0.820	0.826	0.847	0.834	0.920	0.890	0.882	0.882	0.792	0.742	0.715	0.733
elevators	0.670	0.646	0.623	0.628	0.785	0.691	0.676	0.687	0.598	0.551	0.561	0.539
california	0.843	0.766	0.743	0.785	0.903	0.820	0.804	0.801	0.659	0.612	0.620	0.629
house	0.804	0.753	0.741	0.744	0.787	0.735	0.755	0.726	0.643	0.604	0.600	0.631
average	0.805	0.758	0.750	0.760	0.834	0.778	0.784	0.781	0.680	0.643	0.638	0.646

As it can be seen from Tables 3.1 and 3.2, the average values of retention and $rmse$ are better when the inner evaluation model was the same algorithm as the final predictor. It is also shown in Fig. 3.1 for easier interpretation.

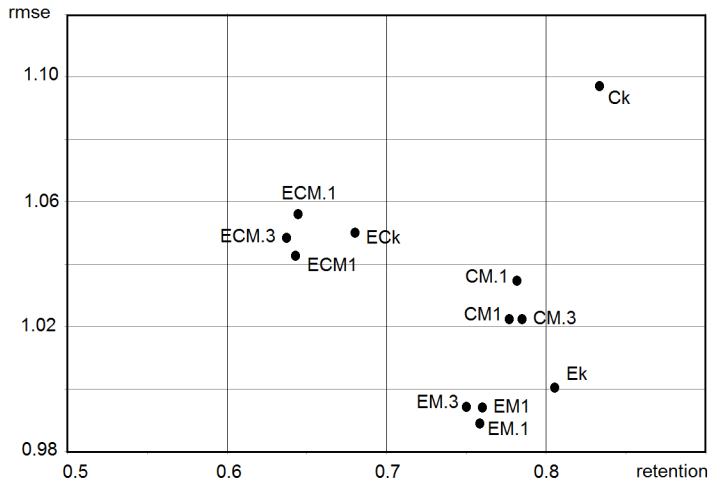


Fig. 3.1. Average retention (lower is better) and $rmse$ (lower is better) for the tested methods over all the datasets.

Fig. 3.2 shows how using the variable Θ parameter (this is $\Theta = \alpha \cdot std$) can improve the instance selection process: in most cases compression is improved a lot, while $rmse$ only a little.

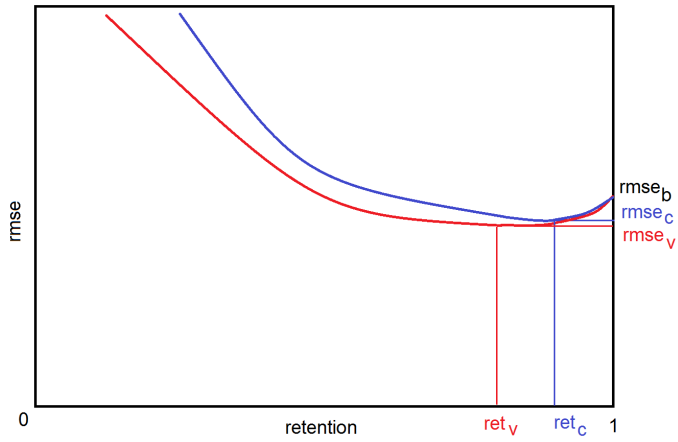


Fig. 3.2. Typical dependence between $rmse$ (on the test set) and retention for instance selection performed with T-ENN + T-CNN . Red: with variable Θ, $\Theta = \alpha \cdot std(Y(\mathbf{X}_S))$. Blue: with constant Θ. $rmse_b$ is the baseline $rmse$ without instance selection.

3.6 Other Solutions from Literature

Definitely fewer papers have addressed the problem of instance selection for regression tasks than for classification tasks. One of the first approaches was presented by Zhang [40], who proposed a method to select the input vectors while calculating the output with k-NN.

Guillen et al. [41] proposed the use of mutual information for instance selection in time series prediction. In the first step the nearest neighbors of a given point were determined and then instead of using k-NN for the prediction, the mutual information between that point and each of its neighbors was calculated. If the loss of mutual information with respect to its neighbors was similar to the instances near the examined instance, this instance was included in the selected dataset. The method was evaluated on artificially generated data with one and two input features. The authors of [42, 43] extended this idea to instance selection in time series prediction by calculating the mutual information between every instance from the training set and the instance be-

ing examined. Then the training set was sorted in descending order by this value and a predefined number of instances were selected.

In [36] a Class Conditional Instance Selection for Regression (CCISR) was proposed and it was derived from the CCIS for classification [23]. CCIS creates two graphs: one for the nearest neighbors of the same class as a given instance and another one for other class instances. A scoring function based on the distances in graphs is applied to evaluate the instances. In CCISR the neighborhood is defined based on the probability density function of the instances, instead of using the nearest instances to construct these graphs.

In [44] an algorithm "to decrease the size of the training set for k-NN regression" (DISKR) was proposed. DISKR first removes the outlier instances and then sorts the remaining instances by the difference of output values between the instances and their nearest neighbors. Next, the instances with little contribution to the training error are successively removed.

In [45] an instance selection method for regression based on recursive data partitioning was presented. The algorithm starts with partitioning the input space using the k-means clustering. If the ratio of the standard deviation to the mean of the group is less than a threshold, the element closest to the mean of each cluster is marked as a representative. Otherwise, the algorithm continues to split the leaf recursively.

In [38] an adaptation of DROP2 and DROP3 to regression tasks was presented and two solutions were proposed: to compare the accumulated error that occurs when an instance is selected and when it is rejected, and to use the same concept of threshold as we presented. Since both ideas were used to adapt DROP2 and DROP3 to regression thus four resultant algorithms were tested. DROP3-RT (Regression-Threshold) worked definitely best of the four methods and thus we used it in comparison with our solutions in the experimental section of chapter 5.

3.7 Conclusions

We discussed the problem of instance selection in regression tasks and presented the threshold-based approach and the discretization-based approach and short review of some other methods. The following conclusions can be drawn from this chapter:

- In classification tasks the data reduction can usually be much stronger than in regression tasks without affecting the prediction capabilities of the model trained on the reduced dataset. That is because in classification we need only to determine the class boundaries and only a few instances are needed for this.
- The threshold-based method usually worked better than the discretization-based one (it will be demonstrated in chapter 5)
- k-NN is much faster algorithm as the inner evaluator than an MLP neural network.

- When the final predictor was an MLP network, the MLP as the inner evaluator produced better results in terms of $rmse$-compression balance than k-NN. As our experiments showed, it also holds true for other models and the optimum is obtained if the inner evaluator is the same model as the final predictor. That is however obtained at the expense of computational time.
- It was usually enough to train the inner evaluator network on 10% of the closest instances. Only for the smallest datasets it deteriorated the results noticeably. This enables much faster training of the network, however, before the training the distance matrix between instances has to be calculated (either exactly or approximately - see chapter 10) to find the closest instances to each given instance.
- Variable Θ threshold in most cases allowed for definitely stronger reduction of the number of instances for a given $rmse$ value. However, the minimum $rmse$ obtained with variable Θ (the maximum noise reduction) was usually only slightly lower than with constant Θ.
- For regression problems the T-ENN algorithm usually produced better results than T-CNN . This is also true for the ensembles of instance selection methods, as will be discussed in chapter 5.
- the CCISR methods [36] and the method of Abdulali [45] use similar concepts to the threshold-based idea with variable Θ.

Chapter 4
Weighting Schemes in Instance Selection

Abstract In instance selection each instance can get either selected or rejected using a crisp decision threshold. However, to improve the results we can introduce instance weighting by assigning each instance a weight reflecting its importance. Moreover, we can use several other weightings while determining the final instance weights, as attribute, diversity, distance and outlier to make the final weight optimally reflect given instance properties.

4.1 Introduction

In a typical instance selection approach we take care about much fewer issues than in typical classification tasks. Let us consider an example of a multilayer perceptron neural network (MLP). This network and instance selection approaches embedded into it are the topic of the third part of the book, but now let us have only a very short look at this model. First the neural network has o lot of weights connecting the neurons. In the network learning process the weights are adjusted to make the network optimally map the input space to the output space. Typically there are hundred of weights for simple problems up to many thousands for complex ones. Owing to this the network has enough parameters to reflect in detail the properties of the underlying data. Moreover, many special algorithms were designed to improve the network learning, like different error functions, weight regularization schema, or incremental architectures that add new neurons when necessary. Each of these measures was designed to deal with particular properties of different datasets and what is worth pointing out, most of the improvements use not binary but real values. Now, when we compare this to instance selection algorithms, they look much simpler. Usually there is only a binary output of the process for each instance and the only properties of the data that the algorithms use are the decision boundaries between classes. Most of the other properties

are ignored in most instance selection algorithms, for example: importance of particular features and instances in the dataset or local properties of the datasets, as diversity and distances between the neighbor instances. In this chapter we discuss the way of including them in the instance selection process. Some approaches to these problems will be discussed in this chapter and some others in the third part of the book.

We present here the weighting scheme for attribute w_{attr}, distance w_{dist}, diversity w_{div} and outlier w_{out} weights. The final weight w, which expresses each point contribution to the prediction model is obtained by a using the four weights. This can be illustrated by the symbolic multiplication of the four weights denoted as "*" in Eq. 4.1. The word "symbolic" means here, that the final weight w is not obtained by the mathematical multiplication, but by considering influence of each weight as discussed below:

$$w = w_{attr} * w_{dist} * w_{div} * w_{out} \tag{4.1}$$

When we use for example an MLP neural network as the inner instance selection model [34], mostly the diversity and outlier weightings are useful, as the network internally performs the remaining operations. However, the prediction results of k-NN depend not only on the value of k, but also on the full weighting scheme [46]. Including all four weights with k-NN as the inner evaluator allowed us to achieve improvement in performance, while making the whole instance selection process much faster than with other models.

4.2 Attribute Weighting

Many classifiers and regressors, as neural networks or decision trees perform internally attribute (feature) weighting, which is one of the core processes of these models. This obviously improves the prediction results, as less important attributes in the data have smaller influence on the final model. Thus this seems reasonable that also instance selection algorithms can take advantage of attribute weighting. As it was discussed in the previous chapter, for the computational efficiency reason the model that performs the inner evaluation is usually k-NN, which does not implement any attribute weighting. If the inner evaluator is a model that performs the attribute weighing internally, then also the instance selection process indirectly uses it. However, in the case ok k-NN evaluator an external method for attribute weighting must be added.

The simplest approach to this is to run one of the feature filters on the training data before running instance selection. Of course feature selection or feature weighting and instance selection influence mutually each other. That will be discussed in the chapter on joint feature and instance selection, but now for the sake of simplicity we can omit this influence, what in most cases will not introduce any noticeable difference (see chapter 6).

However, sometimes we are not sure if a given feature should be rejected and a solution to that problem is feature weighting (more important features are assigned larger weights). Indeed several experiments proved the superiority of feature weighting over feature selection [47]. One of the simplest feature weightings that can be used is to make the feature weights proportional to their correlation with the output. The j-th attribute weight $w_{attr,j}$ equals to that attribute correlation or covariance for standardized data with the output (Eq. 4.2):

$$w_{attr,j} = \frac{1}{N-1}\sum_{i=1}^{N}(x_{ij} - \overline{x}_j)(y_i - \overline{y}) \tag{4.2}$$

where w_j is the covariance between the attribute x_j and the output value y, N is the number of instances (N equals either the number of instances in the dataset or $N = k$ if applied locally to the k nearest neighbors) and b is a coefficient (we use b = 0.5).

In theory the globally optimal set of feature weights may be different than the locally optimal set within the k nearest neighbors of a given point. The global set can be used to find the k nearest neighbors and the locally optimal set of attributes to predict the value of the point of interest. Though it is possible to generate such artificial data that the two sets of weights will be so different that this approach will be unstable, in our experiments on real-world data the differences were small.

We use Eq. 4.3 to calculate the distance $d1$ between two vectors x_1 and x_2:

$$d1(x_1, x_2) = \left(\frac{1}{\sum_{j=1}^{F} w_{attr,j}} \sum_{j=1}^{F} w_{attr,j}(x_{1j} - x_{2j})^p\right)^{\frac{1}{p}} \tag{4.3}$$

where F is the number of features and p is the exponent in the Minkovsky distance measure (for $p = 2$ we have Euclidean distance measure).

4.3 Distance Weighting

Distance weighting is commonly known and used in the so called weighted k-NN and several approaches of that were proposed [48]. The idea of weighted k-NN is that the instances that are closer to an instance of interest should have more influence on the prediction result. We perform distance weighting using the function in Eq. 4.4, where b is a coefficient and $d1(x_1, x_2)$ is the distance between two instances x_1 and x_2 (which can also be the attribute weighted distance from of Eq. 4.3). The function is also shown as the brown line (b=0.25) in Fig. 4.1.

$$w_{dist}(x_1, x_2) = Exp\left(-b(d1(x_1, x_2))^2\right) \tag{4.4}$$

While predicting the output of instance x_1, the output of instance x_2 will be considered with the weight $w_{dist}(x_1, x_2)$.

4.4 Diversity Weighting

Diversity weighting is especially useful for instance selection for regression tasks. It was already implemented and fully analyzed in chapter 3 as the variable Θ threshold, where we proposed to make Θ proportional to the standard deviation of the output of k nearest neighbors of the point of interest $\Theta = \alpha \cdot std\left(Y\left(\mathbf{X}_S\right)\right)$, where α was a coefficient.

4.5 Outlier Weighting

In outlier weighting the instances are assigned weights that express their importance. That is really similar to instance weighting, but when performed for the use of the inner evaluator inside the instance selection process, we call it "outlier weighing" to distinguish this from instance weighting.

The same approach used previously to the final predictor learning can be also applied within the instance selection process. If we want to predict an instance output during the instance selection, then not all neighbors of this instance should have the same influence on the prediction, as some of the neighbors may be outliers and their influence should be limited to same smaller values or even to zero. In this case the instance selection must be run in two passes. In the first pass the outliers are detected and assigned a weight proportional to the probability that a given instance is an outlier, as described in the previous chapter. In the second pass the evaluation model inside the instance selection process predicts the instances output taking into account the weights assigned to their neighbors. Thus the more noisy neighbors have less influence on the decision of selecting or rejecting the current instance in a case of binary instance selection or on assigning a weight to this instance in case of instance weighting. Examples of the outlier weighting functions are shown in Eq. 4.5 and in Fig. 4.1.

$$w_{out} = (Y(\mathbf{x}_i) - \bar{Y}(\mathbf{x}_i))^2 \cdot Exp(-a(Y(\mathbf{x}_i) - \bar{Y}(\mathbf{x}_i))^2) \tag{4.5}$$

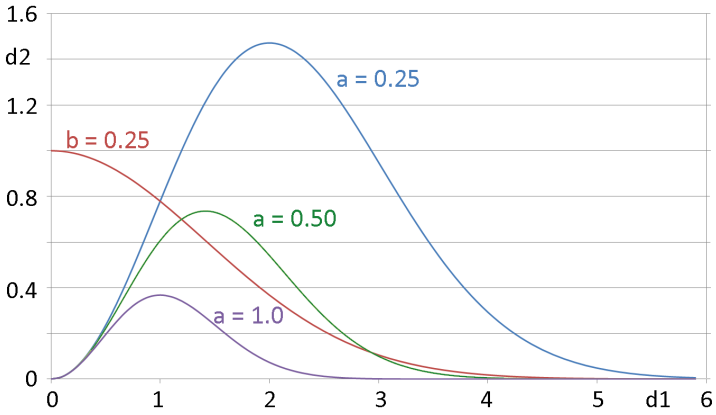


Fig. 4.1. k-NN distance weighting function from Eq. 4.4 (b=0.25) and outlier weighting functions for a = 0.25, 0.50, 1.0.

4.6 Conclusions

In most cases it is a good practice to use feature weighting with k-NN. The improvement due to feature weighting is very significant especially if the final predictive model is different than k-NN. The other weightings added to k-NN (distance and outlier weighting) and to T-ENN (diversity weighting) also improve instance selection results. Outlier weighting with the exponential function (Fig. 4.1) brings some improvement to k-NN and to T-ENN based on k-NN, but the improvement is relatively small. On the other hand, as will be discussed in the third part of the book, the outlier weighting applied in an analogical way to the MLP network error function performs really very well on noisy data, allowing for obtaining much lower prediction errors. Diversity weighting (variable Θ) applied to T-ENN reduces the errors of the final predictor and still more reduces the number of selected instances. In our tests, on average the lowest $rmse$ in the experimental evaluation was obtained when rejecting about 10-15% of instances from the training set without diversity weighting. With diversity weighting the obtained compression for the minimal error was about twice stronger and the number of rejected instanced increased to 20-30% in regression tasks. The final conclusion is that the distance and outlier weighting with exponential functions brings improvement to the T-ENN algorithm, especially when applied to noisy data. When the inner evaluator is k-NN (what is true in most cases), then all the four weights are beneficial. Moreover, as it will be discussed in the third part of the book, outlier weighting allows for significant noise reduction, when implemented into MLP network learning.

Chapter 5
Ensemble Methods in Instance Selection

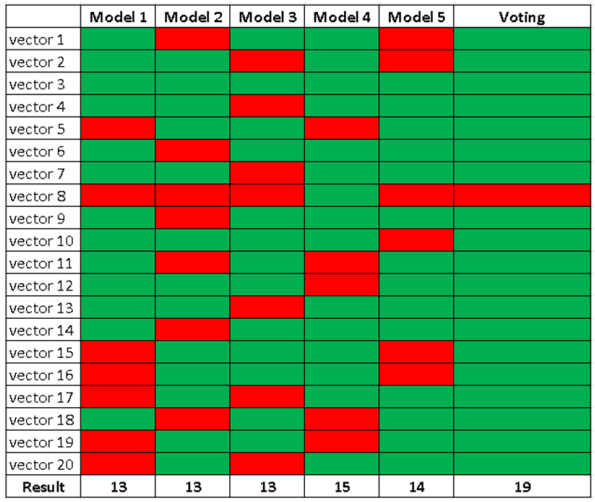
Abstract The idea of ensemble methods is to combine several models (which are either different or learn on different data) to obtain better prediction results than can be obtained by any single model. Each of the models predict the output and then the final output of the ensemble is obtained by averaging the predictions of all single models, what improves the outcome. The same idea can be applied to instance selection with the hope that the results generated by the ensemble will be better than results of any single instance selection algorithm. Moreover, using ensembles allows us to adjust the accuracy-compression balance by using different voting thresholds. In this chapter we present several solutions and discuss in more detail the bagging instance selection ensemble.

5.1 Introduction

The idea of ensemble methods is to combine several models (ensemble members) to obtain higher prediction accuracy or lower $rmse$ than can be obtained by any single model of the ensemble member [49]. Each of the models predict the output and then the final output of the ensemble model is obtained by combining the predictions of all single models.

The idea behind the ensembles is that the combination of several models with lower accuracy (weak-learners) usually allows achieving better results than that of any single model [50]. This is because each member of the ensemble performs poorly in some areas of the data while in other areas it performs very well. Given that poor and good performance areas are different for particular members, combining the predictions through a weighted majority vote (classification) or a weighted average (regression) to produce the final prediction allows for obtaining higher accuracy than that of any single model.

However, to ensure this, each of the ensemble members must be as different from the other members as possible [51]. The diversity can be obtained in two different ways: by using different type of members and/or by training each member on a different subset of the dataset [52]. The later approach is known as bagging [63]. For example, if the prediction accuracy of each single member is 75% and the areas where different members mis-classify the data are different, then averaging their predictions will give much higher accuracy, possibly even 100%. Although in practice achieving 100% in this case will be very difficult, as in some most complex areas of the data the percentage of the ensemble members that miss-classify the examples can exceed 50%, but achieving accuracy over 90% is very likely. The key is here that the members make mistakes for different instances (see Fig. 5.1).



	Model 1	Model 2	Model 3	Model 4	Model 5	Voting
vector 1						
vector 2						
vector 3						
vector 4						
vector 5						
vector 6						
vector 7						
vector 8						
vector 9						
vector 10						
vector 11						
vector 12						
vector 13						
vector 14						
vector 15						
vector 16						
vector 17						
vector 18						
vector 19						
vector 20						
Result	13	13	13	15	14	19

Fig. 5.1. Example of an ensemble classifier. Green color represents a correct prediction for a given instance by a given model and red color a wrong prediction. The last column represents the final prediction of the ensemble obtained by voting of the five models. The result row at the bottom contains the number of correctly predicted instances by each model and by the ensemble.

An important issue is the algorithm used to reach the final decision, so that the correct decisions are more exposed than the incorrect ones. The decision making algorithms can be divided into trainable and non-trainable or into those that apply to class labels and to continuous outputs. An in-depth review of ensemble models for classification is presented in [10], [49], [53] and [54].

Another approach to ensemble learning is boosting, where the instances on which more models fail the prediction are given more attention. In a frequently used boost-

ing algorithm AdaBoost [55, 56] the first base model is trained using equal weights assigned to all instances. In subsequent boosting rounds, the models select the previously misclassified instances with increased probability, making the ensemble to better focus on the difficult cases. However, for very noisy data that is not an optimal approach as the noisy instances are more frequently mis-classified and boosting their role in the learning process may lead to the situation that the ensemble also learns to represent the noise in the data.

An example of ensemble method is random forests [57], which consists of decision trees, where each tree is created using a sample drawn randomly with replacement (so called bootstrap sample) from the training set. So each tree learns on a different subset of instances. In addition, each tree can learn also on o different random subset of features to further differentiate the ensemble members.

Stacking [58] is an ensemble learning technique that combines multiple classification or regression models via a meta-classifier or a meta-regressor. The base level models are trained on the training set, then the meta-model is trained on the outputs of the base level model as features or jointly on the original features and on the base model outputs.

Ensemble models find a broad application in machine learning. An AdaBoost ensemble of neuro-fuzzy classifiers was presented in [59] and [60] including the combinations of fuzzy rules generated by different systems. The ensemble methods were applied also in deep learning [61], including unweighted averaging and majority voting, the Bayes Optimal Classifier, and the Super Learner, for image recognition tasks. The use of ensemble models for time series prediction was presented in [62] and several other papers.

5.2 Bagging of Instance Selection Algorithms

To discus different aspects of ensemble methods in instance selection, in this chapter we will use instance selection bagging ensembles. However, much of the presented information is also valid for other instance selection ensemble methods.

The idea of bagging was presented in the previous section. Although in bagging ensembles the final decision is made by the voting of the ensemble members, where the vote of each member is equally important, it is not said that for the instance to be selected it must accumulate at least 50% of votes, but the selection threshold can be a parameter of the bagging ensemble.

We present some experimental results using the bagging instance selection ensemble for classification and regression tasks [66, 38]. The results for regression tasks will be presented with more details, as this is a more complex problem and the other applications of ensemble methods listed in the previous section were designed for classification tasks.

Here each individual instance selection algorithm (each member of the ensemble) gives one vote for each instance in the subset. In case of binary voting a positive vote indicates that the instance has been selected by the algorithm. In case of real-value instance weighting, each individual instance selection algorithm returns an array a real numbers (votes) between 0 and 1 reflecting the importance of particular instances.

Then all the votes that a given instance received from each algorithm are summed. The importance of an instance in the training set is considered proportional to the sum of accumulated votes. The bagging of instance selection algorithms performs thus real-value instance weighting. For that reason even if we want to achieve the instance weights rather than only the reject/accept decisions, the member algorithms can still perform binary instance selection.

To make the final decision about each instance selection, in a case of binary selection an accept/reject threshold is defined as a percentage of votes an instance must accumulate to be included in the final training dataset. Modifying the parameters of the model we can adjust the trade-off between the prediction accuracy and the dataset compression. This approach can be used as well for instance selection for classification as for regression tasks. In case of the real-value instance weighting, the weight assigned to each instance is expressed by the percentage of the member algorithms that voted for that instance.

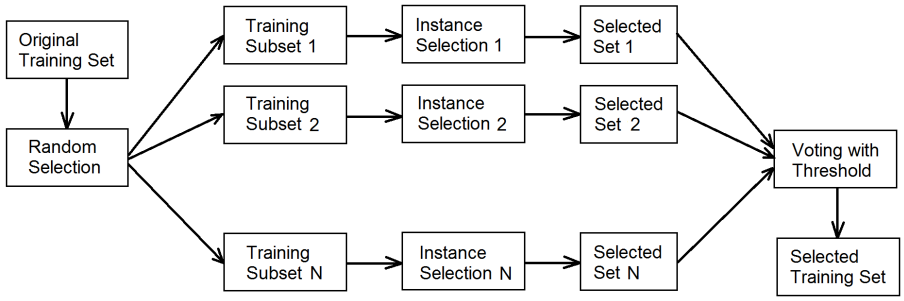


Fig. 5.2. The process of instance selection bagging.

Thus one advantage of ensembles in instance selection is already obvious: the possibility to obtain different trade-offs between the number of instances and the prediction accuracy of a model trained on the reduced dataset. Another advantage is that in a single instance selection algorithm the decision which of two close instances should be removed frequently depends on the order in which the instances are considered, which can lead to suboptimal decisions. Instance selection ensembles can greatly reduce that problem. The final advantage is that both of the criteria: data compression and prediction quality can be frequently better addressed by the ensemble than by a single instance selection method.

Although, in regression tasks using the threshold-based approach (section 3.2) it is possible to adjust the error-compression balance also without ensemble methods, using the ensembles allows obtaining better Pareto-front (lower error and stronger compression at the same time), as will be shown in the next section. Pareto front is set of such solutions that for any of the solution no other solution exists that achieves better values of all the objectives, as shown in Fig. 7.3.

5.3 Experimental Evaluation of Instance Selection Bagging Ensembles for Classification Tasks

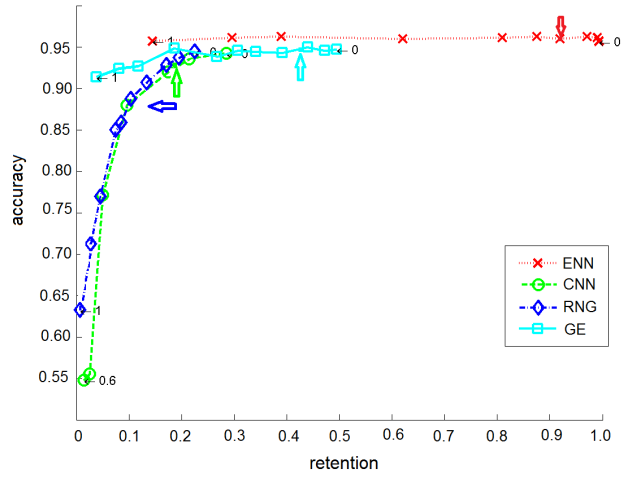
We conducted experiments to assess how particular instance selection methods perform in terms of the prediction error obtained at a given compression level and to find out which methods should be used in which situations. We performed the experiments on all the datasets from the Keel repository [8], which are shown in Tables 1.3 and 1.4. Here, because of space limitations, we show the detailed results only for two datasets in Fig. 5.3.

As it was previously discussed, for classification tasks the instance selection ensembles bring an important advantage of the possibility of adjusting the accuracy-compression trade-off, what is not achievable with single instance selection algorithms, with the exception of very few methods (as the Monte Carlo or RHMC - see chapter 1), where we define the number of selected instances. Thus, when in this section we provide experimental results of the bagging instance selection ensembles used for classification tasks, we will not use the same Pareto Fronts as in regression tasks, because that cannot be obtained for a single instance selection algorithm. Instead we will show how the solutions obtained with different acceptance rates of the ensemble compare to that of a single algorithm. The comparison will allow us to assess how particular instance selection methods perform in terms of the prediction error obtained at a given compression level and thus to be able to take better decisions about which methods should be used in which situations.

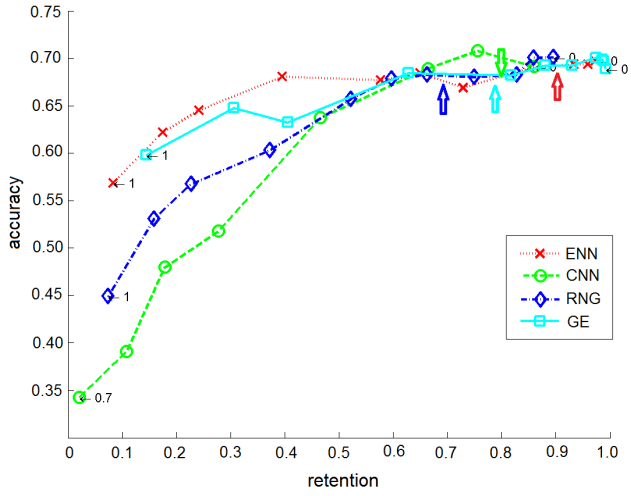
For classification tasks we present selected experimental results for the following instance selection algorithms, which were presented in chapter 2:

- ENN
- CNN
- RNG
- GE

We used the same experimental setup as shown in Fig. 5.2 for instance selection ensembles for classification and regression tasks, only with different member algorithms.



(a) WBC



(b) vehicle

Fig. 5.3. Classification accuracy and retention for two selected datasets. Lower values on both horizontal and higher on vertical axis are better, as this means higher classification accuracy on the test set and stronger compression of the training set ($compression = 1 - retention$). Each point represents one selected training set obtained with different voting threshold. The points are connected with lines for each instance selection algorithm to make them easier to analyze. The arrows show the results of a single instance selection algorithm.

5.4 Experimental Evaluation of Instance Selection Bagging Ensembles for Regression Tasks

In this section we provide experimental results of the bagging instance selection ensembles used for regression tasks. Some of the results were originally published in our work [38].

In instance selection for regression tasks, even a single instance selection algorithm can give as several solutions along the Pareto front, by changing the threshold Θ or the parameter α in Eq. 3.2. For that reason we present the results in the form of Pareto fronts for each algorithm.

For regression tasks we present selected experimental results for the following instance selection algorithms, which were presented in chapter 3:

- T-ENN : threshold-based ENN.
- T-CNN : threshold-based CNN.
- D-ENN: discretization-based ENN.
- D-CNN: discretization-based CNN.
- TE-ENN: ensemble of threshold-based ENN.
- TE-CNN: ensemble of threshold-based CNN.
- DE-ENN: ensemble of discretization-based ENN.
- DE-CNN: ensemble of discretization-based CNN.

From the experimental evaluation, we obtained Pareto fronts for each dataset and each instance selection method. In the experiments we were modifying various parameters of the processes (e.g. the number of discretization bins, or the value of α for calculating the threshold) to see how this influences the instance selection results.

The trade-off between $rmse$ and compression for particular points was adjusted by some parameters of the instance selection methods or by the number of ensemble members that had to vote for a given instance to make it finally selected. Ranking of the methods over all the datasets allows to draw conclusions about the properties and the efficiency of each evaluated instance selection method.

The discretization method and the threshold method of instance selection for regression tasks used in the ensembles were described in chapter 3. Because the inner evaluation algorithm was k-NN, so the process must start from input (attribute) standardization to obtain reliable calculation of the distances between instances. The results presented here were obtained in a 10-fold cross-validation. The testing part of the process contained a bagging inside, and each time one of the evaluated instance selection algorithms was used to build the bagging ensemble. In the the testing part for non-ensemble methods, one of the instance selection algorithms was directly used.

The bagging ensembles used in our experiments consisted of 30 members, each of them was an instance selection algorithm which operated on a different subset of the training dataset. Each subset was created by randomly drawing instances without replacement from the training set. The number of instances in the subset was 80%

of the instances in the training set. An instance was finally accepted if at least $z\%$ of the bagging members accepted the instance, otherwise it was rejected. Changing the z parameter, we can change the behavior of the ensemble so that it prefers rather small resultant datasets (high compression), when z is close to 10%, or it prefers high prediction accuracy, for higher values of z, as 90%.

The experiments were performed with various values of the following parameters of the algorithms:

- Threshold controlled by α: from 0.1 to 1 in steps of 0.1 (for threshold-based instance selection).
- Maximum number of bins: from 5 to 15 in steps of 1 (for discretization-based instance selection).
- Percentage of votes to select an instance z: from 10% to 90% in steps of 10% for the ensembles.

After the instances were selected, they were used as the training set to predict the output of the test set using k-NN with different k values. Here we present the results with the optimal k, including the Pareto front obtained for several representative datasets. Horizontal axes show the retention rate with larger values representing weaker compression. Vertical axes show prediction $rmse$, with larger values representing higher error. $Baseline$ indicates the $rmse$ obtained without instance selection with the optimal k. As in the experiments presented in previous chapters, it was most much easier to improve the results where the final model was 1-NN.

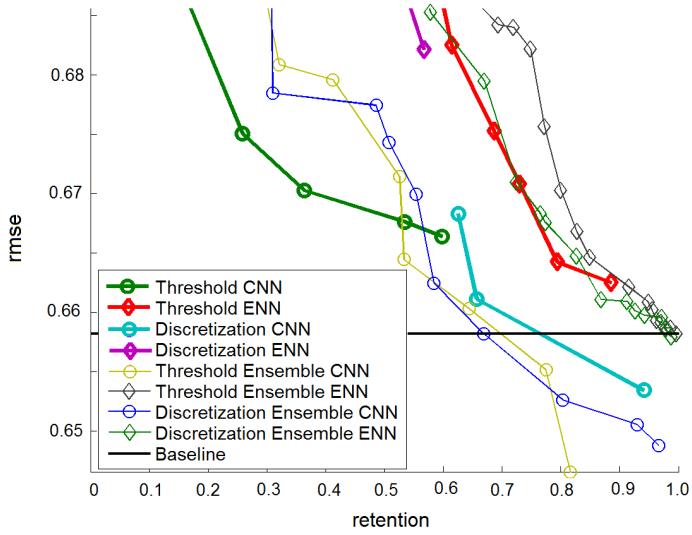
As it was already discussed, we cannot expect such a strong compression, as in classification tasks, but we can at least expect that the ensemble methods will produce better position on a Pareto front than the single instance selection algorithms, and as the results showed it is really the case.

As it can be seen in Fig. 5.4 and 5.5, the T-CNN and D-CNN algorithms in regression problems could successfully remove usually no more than 20% of the instances without causing a noticeable error increase. In the case of T-ENN and D-ENN, which remove outliers, the reduction rate in classification and regression can be comparable, because it depends on the amount of noise in the data and not on the prediction task. Thus on average in classification CNN removes more instances than ENN, while in regression, there is no such a rule and it strongly depends on the properties of particular datasets.

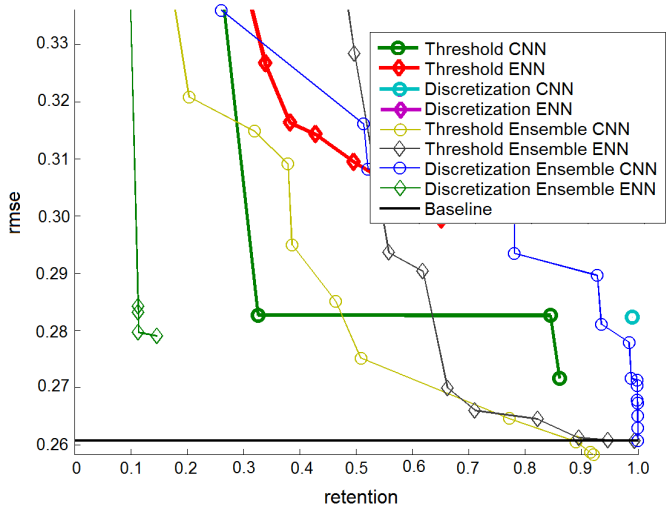
Fig. 5.6 and Table 5.1 show the average results over all the datasets as the value of the objective fitness function F:

$$F = c \cdot R^2 + (1 - c) \cdot compression \tag{5.1}$$

for particular c, where c is a parameter ($c \in [0, 1]$) that allows trade-off balancing between data size reduction and prediction error. The objective function F should take as high values as possible. Its both components: R^2 and $compression$ can take values between 0 and 1. The reason that we use here the coefficient of determination

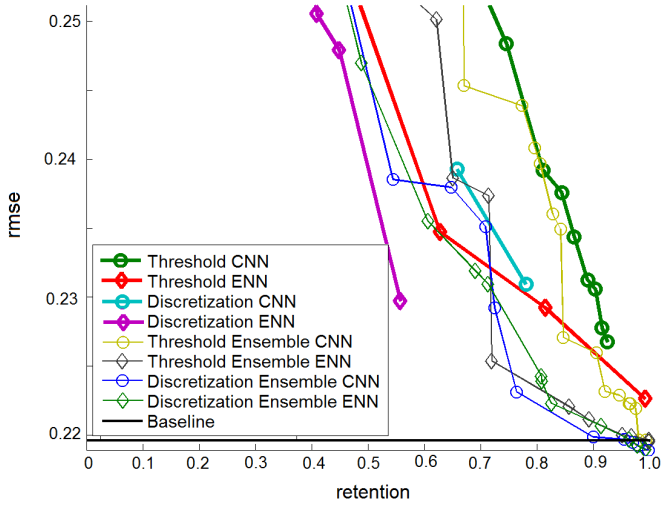


(a) ailerons

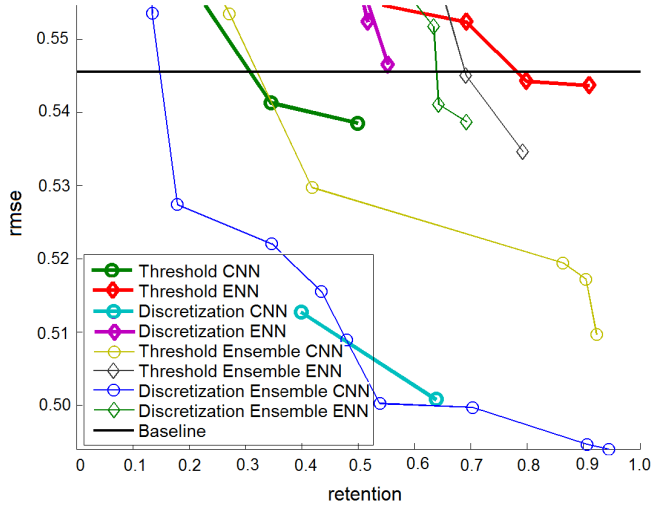


(b) compactiv

Fig. 5.4. $rmse$ and retention for two selected datasets. Lower values on both axes are better, as this means lower $rmse$ on the test set and stronger compression on the training set ($compression = 1 - retention$). Each point represents the result of one set of parameters. The points are connected with lines for each instance selection algorithm to make them easier to analyze.



(a) wankara



(b) dee

Fig. 5.5. $rmse$ and retention for two selected datasets. Lower values on both axes are better, as this means lower $rmse$ on the test set and stronger compression on the training set ($compression = 1 - retention$). Each point represents the result of one set of parameters. The points are connected with lines for each instance selection algorithm to make them easier to analyze.

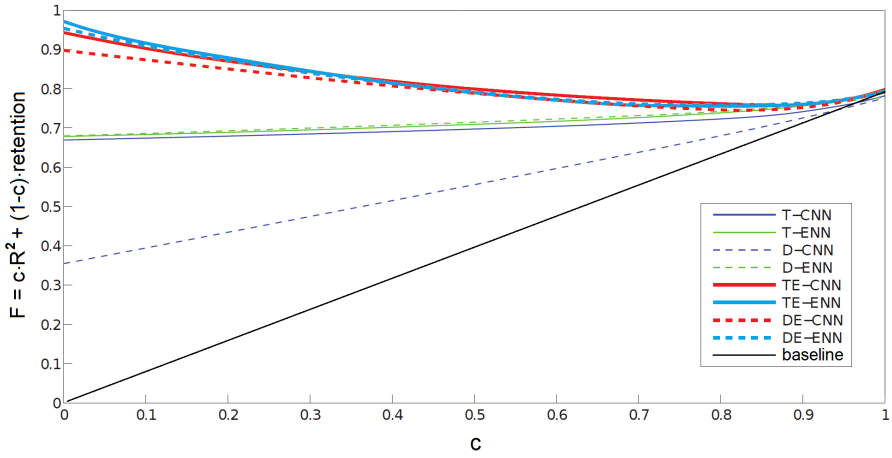


Fig. 5.6. Comparison of the instance selection algorithms: average value over all tested datasets of the criterion function $F = c/rmse + (1-c) \cdot retention$ for different values of the parameter c. (source: our work [38].)

R^2 and not $rmse$ is because R^2 is limited to the interval between 0 and 1, and thus is easier to compare the results over many datasets in ranking systems.

Table 5.1. Comparison of ensemble vs corresponding non-ensemble instance selection methods; number of times a given performed better by achieving higher score in the objective function $F = c \cdot R^2 + (1-c) \cdot compression$ for a given c parameter.

Algorithm	c=0.3	c=0.5	c=0.7	c=0.8	c=0.9	c=1.0
TE-ENN vs. T-ENN	22 / 4	18 / 8	17 / 9	16 / 10	13 / 13	22 / 4
TE-CNN vs. T-CNN	26 / 0	25 / 1	25 / 1	23 / 3	21 / 5	25 / 1
DE-ENN vs. D-ENN	22 / 4	22 / 4	15 / 11	15 / 11	16 / 10	24 / 2
DE-CNN vs. D-CNN	26 / 0	25 / 1	25 / 1	24 / 2	24 / 2	25 / 1

Fig. 5.6 shows the ranking of all algorithms based on their objective (fitness) function for c values within the interval $[0, 1]$. Fig. 5.6 shows that on average the ensemble methods performed better than the individual methods for all the c values. What is also interesting here, that for some datasets very strong compression can be obtained without increasing the error.

5.5 Other Solutions from Literature

Instance selection can be considered a two-class classification problem, where the two classes are "select" and "reject". Thus the idea of ensemble methods can also be applied here. However, the difference is that in instance selection we do not now directly the proper decision (so we cannot check the results in cross-validation process). We can check how a given subset of selected instances influences prediction of a particular model. Several ensemble methods were proposed for instance selection. First we will shortly review these solutions and then we will discuss the problem in-depth using the bagging instance selection ensemble. Below we present examples of other instance selection ensemble methods that can be found in literature.

Feature Bagging. Feature Bagging works in a similar way to bagging. However, here each ensemble members is provided with all the instances of the training set, but a with random subset of features [64]. Also here the decision about each instance selection/rejection is made by the equal weight voting of the ensemble members and also here the number of votes an instance has to accumulate to be selected can be either 50% of votes, or a user-defined parameter.

Boosting. Boosting starts in a similar way as bagging, by providing random subsets of the training set to particular ensemble members. However, when each instance class is being predicted by the inner evaluation model withing the instance selection, the rate of misclassification is recorded. Then the next round of the algorithm is run and the misclassified instances are more likely to be drawn. Additionally the vote of each member comes to the final decision with a weight proportional to this member accuracy. In [29] several boosting methods, as AdaBoost, FloatBoost, MultiBoost and ReweightBoost, were applied for instance selection.

Additive Noise. Additive Noise was proposed by Marcin Blachnik in [64] where each set of selected instances is obtained using a base instance selection method and the dataset for each ensemble member is generated from the training set $\mathbf{T}$ by adding noise to each input instance. All member votes are equally important. The method is partially based on a similar concept, which was first proposed for classifier ensembles.

Other Ensembles Methods. In [65] an instance selection ensemble that used democratic vote of classifiers was presented. In this solution, the instances which were most frequently misclassified get rejected. Instance selection can also be used to create an ensemble of classifiers, where, instead of using bagging, or boosting, various instance selection algorithms provide the diversity of data used for training the classifiers.

5.6 Conclusions

We discussed bagging ensembles of instance selection methods form classification and regression tasks. The following main conclusions from the study can be formulated:

- The ensemble methods allow for balancing the accuracy-compression trade-off by undemocratic voting, even if a single instance selection algorithm used within the ensemble does not allow for this. In regression this was also true, but not always so crucial, as the single algorithm could also sometimes adjust the trade-off.
- In classification tasks we were not always able to obtain with an ensemble stronger compression and higher accuracy at the same time, however in some cases we were able to improve these two objectives more than the single algorithm.
- In regression problems the ensemble performed on average better than in classification in that respect and in most cases we were able to obtain with an ensemble stronger compression and lower $rmse$ at the same time (see Fig. 5.6).
- A big advantage of using instance selection ensembles for both classification and regression is that we do not have to carry so much about the algorithm parameters, as an ensemble members do not have to be so well optimized as a standalone algorithm, as in the process of averaging the answers of particular members the ensemble will do its job to achieve the best results, as long as the members are significantly different.

Chapter 6
Joint Feature and Instance Selection

Abstract There are two dimensions of data reduction: selection of features and of instances. Each of the selections influences the result of the other selection. In this chapter we discuss various options of performing the selections and try to answer the question about the best way to performing both of them depending on the data properties.

6.1 Introduction

Data selection can be performed as feature selection, instance selection, or joint feature and instance selection. The last option can seem the best as it examines both possibilities and combines them into the final result. Thus, in this chapter we will present and discuss some solutions of joint feature and instance selection using similarity-based methods.

Much more research so far was concentrated on feature selection than on instance selection and therefore detailed discussion of feature selection methods can be easily found in literature [67, 68]. As this book is focused on instance selection, we only shortly outline the bases of feature selection to such an extent that is needed to understand how feature selection can coexist with instance selection.

The approaches to feature selection and instance selection are different. For example: a single feature can be assigned some predictive power, while it does not make much sense to assign predictive power to individual instances. Another example are wrappers with forward feature selection, which is a reasonable solution, when the number of features is not so high and when the first feature is selected based on its predictive power. In instance selection, this approach is not commonly used, because of the problem with determining the predictive power and because there are usually much more instances than features and the wrappers methods would be very costly.

The exception, as will be discussed in the second part of the book, is the evolutionary data selection, where although still some differences exist, instance and feature selection can use the same framework.

First we will shortly outline the idea of the three main families of feature selection methods: filters, wrappers and embedded methods. Then we will present some solutions of joint feature and instance selection with similarity-based methods.

The evolutionary methods of feature selection will be discussed in Part 2 and the embedded methods in Part 3 of the book in the context of joint feature and instance selection.

When the expert knowledge is available it can of course be used to make some preliminary feature selection and then the set of features selected by an expert can be refined by feature selection methods [69, 70]. It is especially beneficial if the expert can select the feature set that is commonly understood to specialists in their domain. However, in this chapter, we do not use the expert's knowledge, because it would be understood only by specialists from the given domain.

6.2 Feature Filters

Feature filters use statistical measures to assign weights to individual features. This allows us to reject features with the lowest weights.

A feature ranking created with the help of a filter is not always optimal because it has not been verified in practice, but its advantage is simplicity and speed. This is especially important with huge data sets, where we first need to reduce the dimensionality so that the data can be further processed.

Some ot the feature filters that can be used for classification are: Information Gain, Information Gain Ratio, Symmetrical Uncertainty, Chi-square, Inconsistency Criterion, fast correlation based filter (FCBF), Fisher Score, Spectral Feature Selection, Laplacian Score (LS), Chi squared and filters based on entropy [71].

Some of the feature filters can be used for classification and regression: Minimum Redundancy, Maximum Relevance, Correlation and Relief.

Feature Filter Example - Correlation Coefficient

After the ranking, we can often remove features that are strongly correlated with other features. An example is shown in Fig. 6.1 on the left, where the f1 feature is 100% correlated with the f2 feature, but the best prediction (red or green class) gives the sum of both of these features f1 + f2 (the dividing line is inclined at 45 degree to each axis).

$$correlation(f, y) = \frac{cov(f, y)}{\sigma_f \sigma_y} \tag{6.1}$$

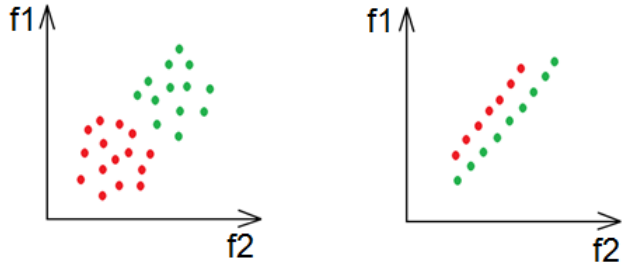


Fig. 6.1. Explanation to the text about feature ranking and correlation. Colors represent classes.

$$covariance(f,y) = \frac{1}{N-1}\sum_{i=1}^{N}(f_i - \bar{f})(y_i - \bar{y}) \tag{6.2}$$

where f is the examined feature and y is the output variable. The filter based on correlation coefficients has the disadvantage that it detects only linear relationships between a given input and output. One method, especially useful in regression problems, to circumvent this problem is to apply a nonlinear matching of the input to the output through various input transformations, e.g. squaring, root, logarithm, partial approximation, etc. and then calculating the correlation coefficient between the transformed input and output. An alternative is to use Spearman correlation (6.3). Spearman correlation of 1 means that the feature and output value are monotonically related, even if their relationship is not linear.

$$spr(f,y) = \frac{\sum_{i=1}^{N}(f_i - \bar{f})(y_i - \bar{y})}{\sum_{i=1}^{N}(f_i - \bar{f})^2 \sum_{i=1}^{N}(y_i - \bar{y})^2} \tag{6.3}$$

Feature Filter Example - Information Gain

The information gain criterion IG is defined as the difference between the entropy before and after the optimal data split on feature f:

$$IG_f = -\sum_{i=1}^{C} p(c_i) \cdot log(p(c_i)) + \sum_{b=1}^{B}[\frac{N_b}{N}\sum_{i=1}^{C} p(c_{bi}) \cdot log(p(c_{bi}))] \tag{6.4}$$

where $p(c_i)$ is the probability that an instance belongs to class i and $p(c_{bi})$ is the probability that an instance within the bin b belongs to class i. N is the number of all instances and N_b is the number of instances in bin b, C is the number of classes and B is the number of bins.

Discussion

A disadvantage of ranking methods is that they do not take into account the interdependencies between features. Sometimes features that have a low correlation with the output can significantly improve the prediction if they are added to another feature. An example is provided in Fig. 6.1.

Fig. 6.1 shows some quite common occurrence: generating new features by a linear combination of existing features (eg f3 = a * f1 + b * f2, here: a = 1 and b = 1), we can effectively reduce the dimensionality of the data set. One of the most commonly used methods is Principal Component Analysis (PCA), which generates new features from existing ones and sorts them according to the amount of information contained in a given feature. Then we can leave only the most informative of the transformed features, rejecting the other ones. It often works very well. The disadvantage of this approach is the fact that new features are difficult to interpret logically, e.g. to interpret the new feature f = 0.75 * voltage + 0.34 * temperature - 0.25 * pressure. So if we need to be able to easily understand such a system and simply draw logical rules out of it, then this is not the recommended way to go.

Experimental Evaluation

The purpose of this evaluation was to select the feature filters that will be used together with instance selection algorithms to reduce the dataset size. We considered two criteria: classification accuracy for a given percentage of selected features and execution time of a given method.

We tested the following feature filters available in RapidMiner: Information Gain, Information Gain Ratio, Deviation, Chi Squared, Gini Index, Uncertainty, SVM, PCA, Correlation, Rule, Relief and three wrappers: forward selection, backward selection and evolutionary selection. Although the backward selection wrapper and evolutionary selection were able to discover more informative feature subsets, resulting in a bit higher classification accuracy with the same number of features, their execution time was between two and four orders of magnitude longer, what in the case of the biggest data sets was totally impractical for our purposes. The results of the filter evaluation are presented in Table 6.1 in terms of the average classification accuracy over the classification datasets obtained in 10-fold cross-validation and the average relative calculation time. The evaluation was done for a predefined number of features equal to the nearest integer to 60% and 30% of the original number of features. Based on the test results, the SVM-based filter produced the best accuracy, but for the further experiments we choose the second filter in the accuracy ranking: Information Gain, because the SVM-based filter was over 100 times slower.

Table 6.1. Average values over the 10 datasets of classification accuracy of neural networks for the nearest integer of 60% and 30% of features (F60-acc, F30-acc) and execution time relative to Information Gain time with different feature filter methods using the RapidMiner implementation.

Method	F60-acc	F30-acc	time
no selection	92.74	92.74	0.0
Information Gain	92.12	91.02	1.0
Information Gain Ratio	92.37	89.80	1.0
Deviation	91.78	88.37	0.2
Chi Squared	91.82	90.48	0.8
Gini Index	92.07	89.52	1.1
Uncertainty	91.82	91.04	1.9
SVM	93.01	91.24	102
PCA	92.51	89.13	0.5
Correlation	89.35	87.40	0.1
Relief	93.02	88.27	245
Rule	92.15	88.44	16

6.3 Feature Wrappers

To remedy the mentioned imperfections of filters, we can use wrappers, although they have a higher computational effort. Wrappers work together with the learning model. They use subset of features to train the prediction model. Then the accuracy of the model is checked on the test set. Next, based on these results, another subset of features is tested and the accuracy is recorded again and so on. It is an optimization task with two criteria: maximization of the prediction quality (classification, regression, grouping) and minimization of the number of features. We can attach a certain weight to each of these criteria. In the extreme case, there may be a zero weight assigned to the number of features. In this way, the selection of features will only be made to increase the accuracy of the model. The number K of possible combinations of F features is given by the following equation:

$$K = \sum_{n=1}^{F} \left(\frac{F}{n} \right) = \sum_{n=1}^{F} \left(\frac{F!}{n!(F-n)!} \right) \tag{6.5}$$

For example for F = 10: K = 1.0E3, for F = 30: K = 1.1E9 for F = 70: K = 1.2E21, for F = 170: K = 1.5E51. With a number of features above a dozen or so, searching the entire solution space is practically impossible. Therefore, various search methods are used, such as best-first search, beam-search, random hill-climbing and even genetic algorithms.

For local search based feature wrappers, forward selection and backwards selection are commonly used. In the first case, we start with a single feature and add further features that most improve the quality of the model prediction. In the second case, we start from the full original set of features and remove further features, starting from theses ones, which if getting removed, will improve the results most. If we care more about the reduction of features, we can also remove more features even if their removal worsens the prediction results, but we still remove the features in this order, which least worsens the prediction results.

Often, the forward selection gives better outcome: greater accuracy of prediction with fewer features. But not always, because in some cases it does not find the best configuration if it is expressed by several features, where each of them has weak predictive properties, but the whole group the features have quite high predictive power.

On the contrary, in instance selection the wrapper approach usually does not make sense, because there are in most cases much more instances than features and thus the process would be extremely time consuming.

6.4 Joined Feature and Instance Selection

The purpose of this analysis is to find the optimal way of using feature selection (FS) together with instance selection (IS). As there are many feature selection methods and many instance selection algorithms it is impossible to test all possible combinations. For that reason we decided first to choose one (in our opinion the best) feature selection method and one (in our opinion the best) instance selection algorithm and test various possibilities of joining them together for optimal outcome.

Our first intuitive approach was to start from the entire dataset and then reduce iteratively one feature and several instances then the next feature and several instances and so on. We tested also many other configurations, such as FS-IS, IS-FS, FS1-IS1-FS2-IS2, FS1-IS-FS2 and others.

However, the results that we obtained showed something else: noise should be eliminated before data condensation. So the first step should be to find the noise. It turned out that in most datasets the noise was to a significantly greater degree associated with features than with instances. That is there was higher percentage of noisy or useless features than noisy instances. On general if a useless feature was more frequently noisy than a useless instance (which was more frequently redundant, inside instances of the same class).

Our experiments showed that for typical datasets feature selection (FS) should be performed prior to instance selection (IS). The before mentioned iterative approach did not performed much worse, but was much more complex and had much higher computational cost.

Thus, we can perform efficiently feature selection using all instances, but less efficiently instance selection using all features. Moreover, several feature filters are based

on some measure of correlation or some variants of information gain. Removing too many instances can make them work less efficiently. On the other hand most instance selection methods are based on the distance between the instances. If there are irrelevant features, we may not get the optimal distances. One of solutions to that problem is multiplying the distance component in each feature direction by this feature importance obtained from some feature ranking (see chapter 4).

The general rule that noise should be removed first and redundancy next is valid as well for instance selection only as for joint feature and instance selection. That however depends on the sensibility of the instance selection algorithm to irrelevant or noisy features. If the algorithm is based on the assessment of instances with k-NN or other method that uses the instance neighborhood, then the method is very sensible to irrelevant features. However, if the instance selection is based on some other prediction model, which performs inner feature selection (e.g. a neural network) then the wrong order of feature and instance selection is likely to cause less detrimental effect.

There is one more practical issue that using first feature filter to remove irrelevant features and then an instance selection algorithm is faster than doing this the other way. That is because most feature filters are much faster than most instance selection algorithms. For example, let us assume that there are 30 features and 1000 instances in the dataset and that we use the correlation feature filter. Thus we have to calculate 1000·30·1 partial distances in the correlation feature filter and 1000·499·30 partial distances in the instance selection algorithm. Thus performing first feature reduction by 50% gives us in total 1000·30·1 + 1000·499·15 = 7.5M operations, while performing first instance reduction by 50% gives us 1000·499·30 + 500·30·1 = 15M operations.

6.5 Other Solutions from Literature

However, there are also proposition in the literature to use the iterative approach, independently on the data properties. Zhang [72] presented a greedy algorithm to perform simultaneous feature and instance selection. The algorithm started from all the features and instances being selected. Then the sequential optimization iteratively removed the least informative features and instances. Once the least informative feature was obtained, the data matrix was updated by removing the row. Then the least informative instances were obtained and the data matrix was updated by removing the corresponding columns. The process was continued until p features and q instances remained. The authors presented the experimental results only for two datasets (ORL: 40 classes, 400 instances, 1024 features and COIL: 6 classes, 1500 instances, 241 features).

Also the authors of [73] proposed the FIS (Feature and Instance Selection) algorithm, which performs both selections simultaneously and they applied the method to text classification. Their FIS considers a set of documents, classified in one of two classes C and C', which contain a group of words each and operates in two steps. In the first step, it searches for a subset of the original vocabulary that contains the words

that are the best predictors of the given class C. Next, only the documents which contain at least one word from this subset are kept. The second step searches, similarly, in the resultant dataset for a subset of words that are the best predictors of class C'. The output of the FIS algorithm contains the two subsets of features over the resulting documents from the first step.

Souza et. al [74] presented another framework for simultaneous and independent feature and instance selection, which was also based on the idea similar to coevolution in the following steps:

```
Input: Dataset dt, Feature Selection Algorithm fs, Instance Selection Algorithm is,
Evaluation Function ef
Output: Dataset ndt
While(Has Iterations(fs) || Has Iterations(is))
   fsss = Next Solution(fs, dt)
   isss = Next Solution(is, dt)
   eval = Evaluation(ef, dt, fsss, isss)
   Update(fs, eval, fsss)
   Update(is, eval, isss)
EndWhile
ndt = Create Subset(dt, Best Subset(fs), Best Subset(is))
Return ndt
```

Some papers also proposed evolutionary optimization of feature and instance selection [75, 76], but we will discuss this issue separately in chapter 12 in the second part of the book.

6.6 Experimental Evaluation

Experimental evaluation of joint feature and instance selection is presented at the end of chapter 17, together with the approach embedded into neural networks to enable better comparison of the two approaches.

6.7 Conclusions

As instance selection and feature selection operate on different dimensions of the dataset, it is usually a good idea to try both of them together. Although this seems obvious there were not many publications on that topic and among them we did not find a method that really joins these two ideas into a single monolith data selection algorithm. Rather all the solutions considered feature and instance selection separately and rather used some sequential or iterative approach, where the feature and instance selection algorithms operate interchangeably.

Part II
Instance Selection with Evolutionary Methods

Chapter 7
Introduction to Evolutionary Optimization

Abstract In the first part of the book we discussed similarity-based instance selection methods. Their important shortcoming is that we have to define the exact rules of instance acceptance and rejection. Evolutionary algorithms allow us to solve the problem without defining any explicit rules. We just run the evolutionary optimization and the process finds the set of most representative instances. Moreover, the solutions found in this way are frequently better in terms of accuracy-compression or rmse-compression trade-off. In this chapter we introduce genetic algorithms, their variants and operations that will be applied to instance selection problems in the subsequent chapters.

7.1 Introduction

Evolutionary optimization uses an iterative process in which a pool (population) of individual solutions is evaluated. Frequently this process is inspired by natural evolution, but some exceptions can be also found (as FWA [77] or ICA [78]). In evolutionary optimization each individual contains a set of parameters that encodes one solution. The purpose of the optimization is to set the parameters to their optimal values. Each individual solution is assigned the so called fitness value, which expresses its quality. The main step of the optimization relies on iterative modification of the parameters of the individuals (e.g. PSO [79], ICA [78]) or on iterative reproduction and modification of new population basing on parameters from current population (e.g. FWA [77], BA [80], and genetic algorithms, which we will use in the following chapters). Both approaches are designed with a purpose of producing better and better solutions (in terms of fitness values) as the optimization iteratively progresses. The process is continued until a termination criterion is met (usually either the predefined number of

iterations or the predefined fitness of the best individual). Then the parameters of the best individual are used as the solution of the problem.

Evolutionary Algorithms are stochastic algorithms that attempt to solve problems by mimicking the processes of Darwinian evolution using selection and reproduction of candidate solutions. However, there is no clear consensus what the term "Evolutionary Algorithms" represents. One definition says it comprises Genetic Algorithms, characterized by a binary string representation of the candidate solutions, Evolution Strategies, which use vectors of real-value numbers as representation and Genetic Programming. Some uses the term GAEs (genetic or evolutionary algorithms) and some other researcher extend the term "genetic algorithms" also to real-value representation. In this part of the book we will use the term "evolutionary algorithms" to denote the binary genetic algorithms and their extended version with real-value representation.

Evolutionary algorithms can be applied to many optimization problems, where the continuous or at least multi-valued measure of the solution quality (fitness) can be applied. For problems, where the only fitness measure is binary (like decision making problems) they are not more efficient than random search. Fortunately instance selection problems belong to the first group.

In this chapter we provide a very short introduction to genetic algorithms, which is however enough to understand the subsequent chapters on evolutionary instance selection. The good in-depth description of genetic algorithms can be found in several books [1, 81].

7.2 Basics of Genetic Algorithms

In genetic algorithms first we generate a pool of random individuals, where each individual contains a set of parameters that encodes the solution of the problem (e.g. selected instances from the dataset). Then the process iteratively creates new solutions, which are supposed to be better than the previous ones using the selection and crossover operators. Let us demonstrate this using a simple example.

Let us assume that the problem is selection of elements from and 8-element set. Let '1' mean that a given element is selected and '0' that it is not selected. We need also a measure of how good a particular solution is. The measure is called fitness function. Let us assume that the optimal solution is

| 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 |

and that the fitness expresses how many elements are properly selected. Since this is the optimal solution, all elements are properly selected and its fitness is 8. Now let us generate a population of size P=4, which will contain four random individuals and let us calculate their fitness as the number of positions, which are equal to the positions

of the optimal solution:

A: | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | fitness(A) = 6
B: | 1 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | fitness(B) = 6
C: | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 | fitness(C) = 5
D: | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | fitness(D) = 3

Then the genetic algorithm iteratively performs the following operations:

- evaluating the quality (fitness) of the P parents and P children (only parents in the first iteration) and selecting the best P individuals into the next generation
- creating P children: for each child selecting M parents with probability proportional to their fitness function and creating the child by crossover operator
- applying mutation with some probability by randomly changing single positions in the chromosome of randomly selected individuals

Algorithm 7 The genetic algorithm

```
generate initial currentPopulation of P individuals
calculate fitness for currentPopulation individuals
for i=0 ... numIterations do
    apply the crossover operation to generate the newPopulation of P individuals
    calculate fitness for newPopulation individuals
    if optimal solution found or no further progress then
        end process
    end if
    sort together currentPopulation and newPopulation individuals by fitness
    select the best P individuals into currentPopulation
    apply the mutation operator
end for
```

Genetic algorithms are simple to implement, but their behavior is difficult to understand. In particular it is difficult to understand why they frequently succeed at generating solutions of high fitness when applied to practical problems.

The most popular explanation is the building block hypothesis, which says that genetic algorithms perform adaptation by identifying and recombining "building blocks", i.e. low order, low defining-length schemata with above average fitness. Goldberg [1] describes the heuristic as follows: "Short, low order, and highly fit schemata are sampled, recombined [crossed over], and resampled to form strings of potentially higher fitness. In a way, by working with these particular schemata [the building blocks], we have reduced the complexity of our problem; instead of building high-performance strings by trying every conceivable combination, we construct better and

better strings from the best partial solutions of past samplings. Because highly fit schemata of low defining length and low order play such an important role in the action of genetic algorithms, we have already given them a special name: building blocks. Just as a child creates magnificent fortresses through the arrangement of simple blocks of wood, so does a genetic algorithm seek near optimal performance through the juxtaposition of short, low-order, high-performance schemata, or building blocks." Most authors agree with that theory, although a few questioned the role of the building blocks.

The main advantage of genetic algorithms and other evolutionary optimization methods is their ability to exploit the data space in a much broader area than traditional methods based on gradient or search heuristics and thus to find better solutions. Moreover, the reasonably good solution is frequently reached only after evaluating a very little number of potential solutions. For example, if we want to select the optimal instances from a dataset of 1000 instances there will be over 10e300 possible subsets. Genetic algorithm can usually find the best solution or a solution enough close to the best assessing no more than 10e5 possible subsets. Although the solution is not guarantied to be the best possible, for practical purposes it can be so close to the best one that it will make no difference.

7.3 Fitness Function and Selection

There are two basic selection methods: tournament selection and roulette-wheel selection. In tournament selection we select randomly some number c of candidates for each parent and then choose the fittest one from them. In this example we will have two parents and two random candidate for each parent. Let us assume that we have randomly selected the following candidates from the example in the previous section:

A and D for the first parent P1
B and C for the first parent P2

next we choose the fittest candidate for each parent - in this case A (fitness=6) for P1 and B (fitness=6) for P2. If the number of individuals selected for the tournament is larger, less fitted individuals have a smaller chance to be selected, so there is a higher selection pressure, which stronger favors better individuals.

In roulette-wheel selection each individual from the whole population can be selected for a parent with a probability proportional to its fitness function. For this example the sum of the fitness value for the whole population is 6+6+5+3=20. Then we generate a random number r from 1 to 20. If r<=6 then individual A is selected, if 6<=r<=6+6 then B is selected, 6+6<=r<=6+6+5 then C is selected and so on. We

can say that each individual is given a portion of the roulette wheel proportional to its fitness. We spin the wheel and we select this individual on which area the wheel stops.

However, the fitness function can be defined in many other ways, for example:

$$fitness = number\_of\_matching\_positions^{v} \tag{7.1}$$

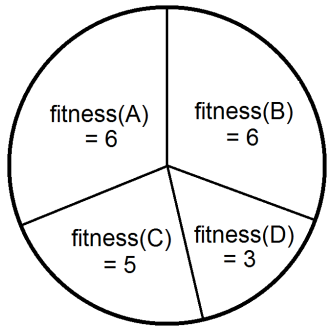


Fig. 7.1. Roulette Wheel Selection. Each individual is given a section of the wheel proportional to its fitness. We spin the wheel and it stop at a random position pointing to the selected individual.

To adjust the selection pressure in roulette wheel selection, we must modify the fitness function. For example, the fitness can be proportional to certain power of the numbers of correct alleles. There exist some correspondence between the number c in tournament selection and the power v in roulette wheel selection (Eq. 7.1). A higher power v roughly corresponds to larger number of candidates c - both displaying stronger preferences towards more fitted individuals.

Sometimes genetic algorithms may converge towards rather local optimum than the global optimum of the problem. If it happens depends not only on the optimized problem but also on the shape of the fitness landscape. In particular it can occur if the convergence of the genetic algorithm is too fast without taking into account other solutions that seem not optimal at a given iteration, but may lead to the global optimum. This limits the population diversity as some potentially good parts of chromosomes included in the overall poor individuals can be removed from the population.

The way to address this problem is to keep diverse population of solutions as long as possible by proper fitness function, selection and mutation. The downside of this is longer optimization and a good balance must be found.

In general, too flat fitness function (which can be e.g. v=1) can make the process very long. On the other hand too steep fitness function (e.g. v=5) may eliminate some less fitted individuals with good chromosome fragments.

Fig. 7.2 shows a typical fitness value of the best individuals and average value of the population in function of the iteration number. As it can be seen, both values change slower and slower as the optimization progresses and both become closer and closer to each other. As the first solution, we can start with low c or low v depending on the selection method in order not to limit the population diversity and when the process progresses further and the differences between particular individuals decrease, we increase c or v in order to accelerate the end of the optimization. Thus, it can be desired to make the selection less steep at the beginning of the optimization to keep the population diversity and prevent converging to a local optimum and steeper at the final stage to strongly promote the (only little) better individuals to accelerate the process.

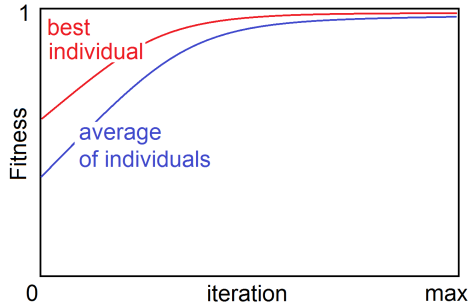


Fig. 7.2. Typical fitness function changes as optimization progresses.

Other possible solution using selection is to apply "niche penalty" or "crowding distance", where, a group of similar individuals (niche radius) have a penalty term subtracted from their fitness value. This reduces the number of individuals of that group in subsequent generations, permitting other, less similar individuals to be promoted to the next generation, even if their original fitness value is lower. Another solution is to replace part of similar individuals by new randomly generated ones. Still another solution is to increase the mutation rate, especially within such a group, to introduce greater diversity.

7.4 Crossover

Using the four individuals from the previous example, we have 4 parents and we want to generate 4 children. To create each child we need to select randomly two or more parents with the probability proportional to their fitness and randomly select crossover points (one or more). In the simplest case we will use two parents and one crossover point. Now the child will be created by taking the first part of its chromosome - up

to the crossover points from the first parent and the second part - from the crossover point till the end from the second parent. Fox example: we randomly selected parent C and A and a crossover point 3:

parent C: | 0 | 1 || 1 | 0 | 1 | 1 | 0 | 0 | fitness(C) = 4
parent A: | 0 | 0 || 0 | 1 | 0 | 1 | 0 | 1 | fitness(A) = 6
child: | 0 | 1 || 0 | 1 | 0 | 1 | 0 | 1 | fitness(child) = 7

We repeat this selection and crossover operators three more times to generate the population of four children. Then we have 8 individuals, from those we choose 4 with the highest fitness, which will enter the next iteration. Because the individuals with higher fitness have higher probability of becoming parents, each next generation is characterized by a highest average fitness. We continue this process until we either find the best solution (if we know the fitness of the best solution) or if we find a solution that we consider good enough or as the process ceases to improve or when the maximum predefined number of iterations is reached.

Two parents and one crossover point in long chromosomes make the optimization unnecessarily too long, as the exchange of genetic material is very slow. Usually more parents and more crossover points are better [82], but using too many may again slow down the process, as it will disturb creating the blocks of well matched alleles in the chromosome.

Algorithm 8 The crossover operation

```
for i=0 . . . P do
  if RandomDouble(0,1) < crossoverProbability then
    for c = 0 . . . numCrossoverPoints + 1 do
      individual[i][c] = Selection();
      crossoverPoint[i][c] = RandomInteger(0,numPositions);
    end for
    sort crossoverPoint[i];
    for c = 1 . . . numCrossoverPoints + 2 do
      for d = crossoverPoint[i][c - 1] . . . crossoverPoint[i][c] do
        newPopulation[i][d] = currentPopulation[individual[i][c - 1]][d];
      end for
    end for
  end if
end for
```

For longer chromosomes there are several issues that must be considered to make the genetic algorithms more effective: the formulation of the fitness function, the size

of the population, the population initialization, the number of parents and crossover points, the mutation operator. That will be discussed in subsequent chapters in relation to genetic algorithm based instance selection.

7.5 Population Size and Initialization

The population size should big enough to ensure the required population diversity, because too small size may result in not finding the solution. Too big size is usually not bad, but the optimization may take too long. For most of the problems considered in the following chapters the size about 100 individuals is optimal. The simplest initialization is random. That is each allele is randomly initialized with either 0 or 1 and in the case of real-value optimization by a real random number between 0 and 1. However, for the purpose of instance selection, more effective initialization schemes exist. We will discuss both issues: population size and initialization while presenting the ways of accelerating genetic algorithm based instance selection.

7.6 Mutation

We can imagine such situation, that the optimal value at some position is 1. However, due to the crossover process after several iteration this value is 0 at each individual. In this case the mutation operator is useful, as it with some probability randomly changes a value at a random position of a random individual. There are cases where the optimization can succeed without the mutation operator. Usually an optimal probability of mutation rates exists: too low probability makes the process too long and too high probability may disturb the order built by crossover operations. Moreover, the mutation rate can be increased gradually during the optimization or unsymmetrical mutation probabilities can be used, where the chance of switching 0 to 1 is different than switching 1 to 0. Some examples applied to instance selection will be presented in the following chapters.

7.7 Elitism and Steady State Genetic Algorithms

There are several options of how we can create the next generation from the child and parent population. The simplest option is that the whole child population is advanced to the next generation and the whole parent population is rejected. It can however happen that some individuals in the parent population are better than any individual

in the child population and thus it would be worth to preserve them. Here we can use elitism, where a percentage of the best parents create the elite and are always promoted to the next generation. Let us assume that 20% of parents are promoted. In this case we need to generate the number of children which is 80% of the population size. Thus that approach can be both: faster and more efficient.

Another option is to sort the parent and child population together and select P (where P is the population size) best individuals to the next generation, no matter if they come from the parent or child population.

Till now we considered only generational genetic algorithms, where the new generation at a certain point was replacing the old one. In a single CPU core environments we can obtain still faster improvement of the population quality if the child replaces one of the individuals as soon as it is created. It was verified [83, 84, 85] that the steady state genetic algorithms display faster convergence than generational ones. The explanation of that fact is that in steady state algorithms, the offspring immediately replaces the worst individual in the population (or the most similar or one of its parent), what at this moment makes the population better as a whole, while in generational genetic algorithms we have to wait for that improvement until the next iteration.

In parallel implementations, not always the solutions with the smallest number of calculations (as steady state algorithms) are fastest, but those that scale up well. We will discuss this in chapter 11.

7.8 CHC Genetic Algorithms

There are a lot of genetic algorithm variants. Here we shortly present the CHC algorithm, as it was applied to instance selection by several authors. In CHC [86] genetic algorithms the parent population is used to generate the intermediate population and then the best individuals from both population enter the next population. CHC uses a different crossover (recombination) operation, so called HUX, which exchanges half of the bits that differ between parents, with the crossover point being randomly chosen. However, if the selected parents are too similar, the recombination is not performed. In CHC the mutation is applied after the recombination phase and the mutation is usually quite strong (up to 35%, comparing with the typical mutation rates of about 1% in classical generational GA).

7.9 Cooperative Coevolution

Cooperative coevolution in evolutionary algorithms is a method, which divides a large problem into smaller subproblems (species) and solves them independently in order to solve the original large problem. The subproblems are solved by independent subpopulations, which interact in the cooperative evaluation of each individual of the subpopulations. The initial fitness of each population member is obtained by combining it with a random members of other populations. In the next and subsequent iterations, the fitness of each individual from a subpopulation is obtained by combining it with the best members of all other populations [87]. The problem with the basic coevolutionary approach is that the algorithm tends to converge too quickly using only a limited search, thus finding the solutions which are not enough close to the global optimum. However, later several improvements to deal with the limitations of basic coevolutionary scheme were introduced [88]. In several works coevolution was used for simultaneous instance and feature selection, where one subpopulation encoded instances and the other features, as will be discussed in chapter 12.

7.10 Multi-Objective Evolutionary Algorithms

So far we have considered single-objective genetic algorithms, where there is only one fitness measure (one objective). For example, in the instance selection problem that can be a weighted sum of data reduction and prediction accuracy. In multi-objective algorithms there are multiply fitness measures, as data compression and maximization of prediction accuracy, which are two independent fitness measures. That allows to search not only for a single best solution but also for a set of solutions, where each of them is the best for a certain balance between the objectives. In multi-objective optimization the aim is in fact to find the Pareto front of non-dominated solutions, what corresponds to a simultaneous optimization with various weights assigned to particular objectives [89, 90].

There are two problems with the single-objective approach: we must know which weights we want to use and if we need several solutions with different weights, then we need to run the optimization several times: each time to get one solution. Multi-objective algorithms allow us not to carry about the weights and obtain a set of solutions corresponding to different weights in a single algorithm run.

For the many problems that can be evaluated in terms of multiple criteria (as instance selection), simultaneous consideration of multiple criteria (objectives) can be done using Multi-Objective approaches. They can be divided into three groups:

1. Scalar approaches. They base on aggregating criteria into a single objective or on processing one objective at the time. In this group of methods it is usually necessary to specify a coefficient that ensures a certain balance between the objectives.

2. Pareto approaches. They base on Pareto front and domination between individuals ([91]). They can be divided into ranking, elitist and diversity maintaining methods [92].
3. Other approaches. They include other methods, such as methods that are based on sub-populations (e.g. VEGA [93]).

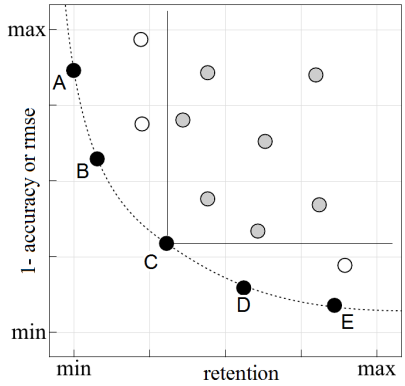


Fig. 7.3. Examples of domination in population: black circles (A, B, C, D, E) are the non-dominated individuals in the population - they are situated on the Pareto front (dotted line). The remaining points are dominated. For example all the gray points are dominated by the point C.

Solutions based on the Pareto front are among the most commonly used. They do not require the determination of the coefficients indicating the expected balance between the objectives. Moreover, the returned result is the front of the non-nominated individuals (Pareto front) with different trade-offs between objectives. We can say that the individual $\mathbf{S_1}$ dominates individual $\mathbf{S_2}$ (with goal of minimization of the objectives) if:

$$\begin{cases} obj_i(\mathbf{S_1}) \leq obj_i(\mathbf{S_2}) & \text{for} \quad \text{all } i \\ obj_i(\mathbf{S_1}) < obj_i(\mathbf{S_2}) & \text{for at least one } i \end{cases} \tag{7.2}$$

where i is the index of objective function ($i = 1, ..., O$), O is the number of objectives, $obj_i(...)$ is the objective function. The examples of domination between individuals and Pareto front can be seen in Fig. 7.3. For example, in the case of instance selection the individuals will be the selected training sets, the first objective will be the error of the prediction and the second objective - the selected data set size.

Several multi-objective evolutionary algorithms have been proposed [94]. The NSGA-II algorithm is the most frequently used and one of the best for two-objective

problems and its extension NSGA-III for problems with more than two objectives [95]. Below we shortly present the main ideas of the NSGA-II algorithm and the interested reader is referred to [94] for more details.

The returned result of the NSGA-II algorithm is the front of the non-nominated individuals (Pareto front) with different trade-offs between objectives. The NSGA-II algorithm is presented in the following pseudo-code:

Algorithm 9 NSGA-II

$\mathbf{Population} := initialization(N)$
$evaluation(\mathbf{Population})$
$\mathbf{Front} = fast\_nondominated\_sort(\mathbf{Pop}, N)$
$crowding\_distance(\mathbf{F})$
while $stop\_condition()$ **do**
  $\mathbf{Population_{child}} = \emptyset$
  for $i = 1$ **to** N **do**
    $parentA = select\_parent(\mathbf{Population})$
    $parentB = select\_parent(\mathbf{Population})$
    $child = new\_individual(parentA, parentB)$
    $\mathbf{Population_{child}} = \mathbf{Population_{child}} \cup child$
  end for
  $evaluation(\mathbf{Population_{child}})$
  $\mathbf{Population} = \mathbf{Population} \cup \mathbf{Population_{child}}$
  $\mathbf{Front} = fast\_nondominated\_sort(\mathbf{Population}, 2P)$
  $crowding\_distance(\mathbf{Front})$
  $\mathbf{Population} = selection(\mathbf{Population}, \mathbf{Front})$
end while
return **Front** {list of non dominated individuals}

Explanation of the above NSGA-II pseudo-code:

1. Initialization of population with P individuals.
2. Evaluation of the population according to the defined objectives.
3. Searching for individual fronts. First, the non-dominated individuals from the population are transferred to the first front. Then, next fronts are selected consequently from remaining individuals. This process performance is optimized by Fast Non-dominated Sort (for the details see [94]).
4. Calculation of crowding distances. Within each front a crowding distance for each individual is calculated (for the details see [94]). It determines the distance between neighboring individuals from a given front and promotes more diverse solutions in a further selection process.

5. Reproduction. In this step a new population with P children individuals is created. For each child a pair of parents is chosen basing on ranking selection (using fronts values and then crowding distance - individuals with smaller values are chosen). The child individuals are created using the crossover and mutation operators.
6. Evaluation of the new population according to the proposed criteria.
7. Merging of the populations. In this step new (child) population and the old (parent) population are merged into one.
8. Searching for individual fronts and calculation of crowding distances for the merged population.
9. Selection. From the population (with size $2P$) P best individuals are selected and individuals from successive fronts are added sequentially to the new population. If the number of individuals of the front is too large (the new population size is higher than P) then only the appropriate number of individuals is selected basing on the crowding distance.
10. Stopping Criterion. In this step a stopping criterion is checked (for example if the number of iterations reached the specified value). If the criterion is met, the algorithm stops and the first front of non-dominated solutions is returned, otherwise the algorithm goes back to the step 5.

Chapter 8
Single-Objective Evolutionary Instance Selection

Abstract In evolutionary instance selection the dataset is encoded into the chromosome, where each position represents one instance. During the optimization the solutions are assessed using a fitness function, which is a weighted sum of the data reduction and of the prediction quality of a model learned on the selected set. To limit the computational cost, we use k-NN as the model that evaluates the prediction quality inside instance selection algorithm, because it is possible to calculate and sort the distance matrices only once before the optimization, what makes the solution very fast. Moreover, several other improvements can be implemented.

8.1 Introduction

Evolutionary algorithms do not make any assumptions about the dataset properties, but verify experimentally large numbers of different subsets in an intelligent way to minimize the search space. This can result in better solutions. On the other hand that is usually achieved at the expense of much higher computational cost. For that reason we pay special attention to limit the computational cost as far as possible, what is discussed in section 8.4 and in chapter 11.

In the case of similarity-based instance selection we discussed separately classification and regression. However, with evolutionary methods we can take a very similar approach to instance selection in classification and regression tasks. The core of the approach can be the same in both cases, only the value of process parameters or some enhancements can differ. For that reason in this and in the next chapter, we will discuss jointly the classification and regression tasks and only point out the differences, where applicable.

8.2 Encoding

The evolutionary instance selection process starts with generating initial random population. Each individual in the population represents one training dataset $\mathbf{T}$. Each position of the individual's chromosome represents a single instance; 0 at this position means this instance is rejected and a positive value - it is selected (1 for binary instance selection and a real number between 0 and 1 for instance weighting). Let us consider an example, where the dataset consists of 10 instances and the first instance is selected, the second rejected, the next three instances are selected, next two rejected and the last three is selected. The representation of this selected subset in the chromosome is:

`|1|0|1|1|1|0|0|1|1|1|`

In instance weighting, real value weights between 0 and 1 are assigned to each instance and each individual (each selected dataset $\mathbf{S}$) is encoded as a weight vector $\mathbf{w} = \{w_1, ..., w_N\}$, where N is the size of the weight vector (which equals the number of instances in the original training set $\mathbf{T}$). The weights express the importance of a given instance while training the classifier or regressor, as discussed in detail in chapter 4 in the first part of the book. Binary instance selection is the simplest form, while on the other hand, implementing instance weighting in regression problems frequently allows for some improvement in prediction quality for noisy data [96, 7].

8.3 The Objectives and Fitness Function

Instance selection is a two-objective task. The first objective is minimization of the number of instances in the training set (data reduction or compression). The second objective is maximization of prediction quality. In case of classification tasks the most popular quality measure of prediction is the percentage of correctly classified instances (the higher the better) and in regression tasks $rmse$ - root mean square error (the lower the better), although other measures are also possible.

During the instance selection process the accuracy on the training set acc_{trn} or $rmse$ on the training set $rmse_{trn}$ is internally determined with the leave-one-out procedure, always using the k-NN algorithm as the inner prediction model (the classifier or regressor inside the instance selection process). In the final evaluation the accuracy on the test set acc_{tst} or $rmse$ on the test set $rmse_{tst}$ can be determined using any prediction model trained on the reduced training set S. In the experimental section the final prediction models we used were k-NN with different k parameters and MLP neural networks and the acc_{tst} and $rmse_{tst}$ obtained on the test sets is reported.

The typical definitions of the fitness function for instance selection are:

$$fitness = \big(\alpha \cdot retention + (1 - \alpha) \cdot (1 - acc_{trn})\big)^{-1} \tag{8.1}$$

$$fitness = \left(\alpha \cdot retention + (1-\alpha) \cdot rmse_{trn}\right)^{-1} \tag{8.2}$$

where α is a coefficient indicating the expected balance between the objectives. Different fitness functions can be used and their influence on the instance selection process will be discussed later. One of the simplest versions for classification tasks is:

$$\mathit{fitness} = \left(\alpha \frac{Accuracy}{avgAccuracy} + (1-\alpha)\frac{avgNumInstances}{numInstances}\right)^{v} \tag{8.3}$$

while for regression it is:

$$\mathit{fitness} = \left(\alpha \frac{avgRmse}{rmse} + (1-\alpha)\frac{avgNumInstances}{numInstances}\right)^{v} \tag{8.4}$$

In Eq. 8.4 $rmse$ is the root mean square error of the model trained on the current training set, $avgRmse$ is the average $rmse$ over the whole population of training sets, $numInstances$ is the number of selected instances in the current training set and $avgNumInstances$ is the average number of selected instances over the whole population of training sets. v is some positive real number, controlling how steep the fitness functions is, that is, how strongly the better solutions are favored. The exponent v makes sense if the roulette-wheel selection is performed, where each parent's likelihood of being chosen for the reproduction (crossover) is proportional to its fitness function. If we use the tournament selection, then the exponent v is irrelevant, as discussed in chapter 1.

In general the first part expresses the prediction quality and the second part data reduction. One point must be clearly stated: as in the case of the similarity-based instance selection algorithms, data reduction is always measured on the training set. However, the quality of the solution can be measured either on the training or on the test set. During the instance selection process we maximize the prediction quality on the training set and the final goal is to achieve the highest prediction quality on the test set. This is in similar way, as we train some model, e.g. a neural network, on the training set and want it to perform well on the test set. And in a similar way some measures to prevent the possible over-fitting must be taken.

8.4 k-NN as the Inner Evaluation Algorithm

The most time consuming part of the genetic algorithm is the evaluation of the fitness function value, as it requires calculating the accuracy or $rmse$ on the training set, so we put a special effort into solving this problem.

Let us assume that there are 96 individuals in the population and that the optimization requires 30 epochs. In this case the value of the fitness function must be calculated

2880 times. Training any prediction model 2880 times would be computationally very costly.

In case of the k-NN algorithm we can calculate the distance matrix between each pair of instances in the training set only once before the optimization starts, and not the mentioned 2880 times. We create one two-dimensional array D_i for each instance x_i. The first dimension is N - the number of instances in the training set and the second dimension is three. The three values stored in the array are: $dist(x_i, x_j)$, j and y_j. $dist(x_i, x_j)$ is the distances between the current instance x_i and each other instance x_j, which in most cases will be an Euclidean distance. The second value j is the ordinal number of each instance in the dataset $\mathbf{T}$. The third value is the output value y_j of the j-th instance.

Then we sort the arrays D_i increasingly by $dist(x_i, x_j)$. At the prediction step we only go through the beginning of the array, read the instance number j and check if this instance is selected, if not, then we go to the next instance, as long as we find k selected instances. Then for the k nearest selected instances we read their weights w_j from the chromosome, their output value y_j and predict this instance output value $y_{predicted\_i}$ as the weighted average of the k outputs (which is a simple average in case of binary instance selection).

$$y_{predicted\_i} = \frac{\sum_{j=1}^{k} y_j w_j}{k \cdot \sum_{j=1}^{k} w_j} \tag{8.5}$$

where y_j is the output value of the j-th neighbor of an instance x_i and w_j is the weight expressing the j-th neighbor importance (it should not be confused with the weight wd_j related to the distance between the instances x_i and x_j as in the standard weighted k-NN algorithm). Only the neighbors with $w_j > \gamma$ are considered in the prediction. In binary instance selection always w_j=1. To prevent an instance to be considered by the algorithm a neighbor of itself, we set the distance to the instance itself to a very large number as 1.79E+308 (max. value of the double type). The prediction step is extremely fast and only this step is performed each time the $rmse_{trn}$ is calculated.

Details of how the weights w_j are determined in similarity-based instance weighting can be found in chapter 4 in the first part of the book. However, using evolutionary algorithms, the weights are determined by the evolutionary process and thus we do not need to explicitly express them with any formulas. However, we may need to use more optimal version of the k-NN evaluator, that is mainly to include attribute weighting in calculations of the distance matrix according to Eq. 4.3.

Although, our previous experiments [34] showed that the best results in terms of $rmse$-compression balance can usually obtained if the inner evaluation model is the same algorithm as the final predictor, using the attribute weighting in k-NN makes the results more similar to those of other algorithms. This we sacrifice only a very small improvement in order to shorten the optimization process usually about three orders of magnitude. In this way we obtain better results, while still keeping the process time short.

In the case of instance selection, we do not have to find the absolutely best result, but an enough good suboptimal solution is acceptable. At the end of genetic optimization the improvement progresses very slowly, so by accepting a good suboptimal solution we can save o lot of time.

A question may arise if the subset of instances that is optimal for this version of k-NN is also optimal for other predictive models. To answer the question we performed experiments, when the model used to evaluate the fitness function was an MLP network with a single hidden layer trained with the Rprop algorithm. The results showed that the subsets selected with k-NN and MLP contained mostly the same points, however some differences can be observed. We assume that even if the subset selected by k-NN is not optimal for MLP, it is enough close to the optimal one and k-NN can be used with pre-calculated distance matrix as much faster fitness function evaluator than an MLP network.

Another question is if we can accelerate the learning of other models in similar way by caching some information. Usually not so efficiently and not so easily as with k-NN. Although when we go from one instance to the next one, the model could incrementally learn the information about the previously considered instance and unlearn the information about the current instance. For models such as a neural network or a decision tree that would be very complex, as it would require storing in some arrays most of the learning history of the model, then making some changes and partially learning the models. Especially for neural networks, which use non-linear transfer functions the gain would be not significant, as most of the model learning would need to be done anew and the complexity of the incremental algorithm would be very high. Thus this approach is impractical.

Thus, we can assume that the proposed solution is the best compromise between the accuracy, speed and simplicity of implementation.

8.5 Experimental Evaluation

In this chapter we present the results obtained for classification tasks. The results obtained for regression tasks are presented in chapter 10 in order to compare them with the results obtained with additional data partitioning.

In Table 8.1 we present the results for k-NN with optimal k and an MLP neural network as the final predictor. The neural networks were trained with the R-prop algorithm. We used networks with one hidden layers. The numbers of neurons in the hidden layer was equal to the geometric mean of number of inputs and number of classes.

The α parameter was used to assign weights to the two objectives: classification accuracy and data reduction. The power exponent v in the fitness function (8.3) was gradually increased during the training. We started from v=2 and finish at v=6, as the

differences between particular individuals tend to get much smaller as the genetic optimization progresses. If an individual contained fewer than four instances, we always added the missing instances from the most frequent instances.

dataset	alpha	ret(IB3)	kNN(IB3)	MLP(IB3)	ret(EV)	kNN(EV)	MLP(EV)
ImgSegm	0.99	13.9	90.0	91.0	13.7	91.4	92.5
	0.96				13.7	91.4	91.6
	0.90				13.7	91.4	91.5
balance	0.99	15.2	78.0	82.3	14.8	83.9	85.1
	0.96				10.0	83.0	84.1
	0.90				5.2	82.4	83.6
Led7digit	0.99	20.1	50.1	50.3	20.3	70.2	70.1
	0.96				16.1	65.0	65.3
	0.90				12.7	62.4	63.3
bupa	0.99	22.8	67.9	68.8	19.6	69.6	70.3
	0.96				14.4	67.9	69.0
	0.90				12.6	65.3	67.5
vehicle	0.99	30.3	62.7	64.1	29.4	64.5	66.1
	0.96				22.0	62.3	63.8
	0.90				16.1	60.7	62.8
penbased	0.99	3.60	97.4	90.0	4.02	97.5	91.3
	0.98				3.20	97.3	91.3
	0.96				2.54	91.5	88.0
PageBlk	0.99	2.29	93.8	93.6	1.86	94.1	97.1
	0.96				1.09	93.3	96.8
	96.75				0.82	92.4	96.5
magic	0.99	3.91	82.9	82.5	2.85	84.9	85.3
	0.96				1.98	84.0	84.5
	0.90				1.15	82.6	83.1
yast	0.99	14.4	48.4	50.2	15.4	52.0	54.9
	0.96				10.1	50.8	51.6
	0.90				6.67	48.4	50.0
twonorm	0.99	3.51	95.3	95.2	2.96	97.5	97.5
	0.96				1.86	95.5	95.4
	0.90				1.12	94.8	94.9

Table 8.1. Experimental results. ret(.) - retention (percentage of selected instances), kNN(.), MLP(.) - classification accuracy obtained with k-NN and MLP neural network as the final predictors for two instance selection methods: ENN+IB3 denoted as IB3 and evolutionary instance selection denoted as EV.

In instance weighting in k-NN each instance was multiplied by its weight and the weighted number of instances in each class was considered. In the MLP network the error the network makes during learning on each instance was multiplied by the instance weight. Instance weighting only seldom improved the results for classification. However, based on our previous experience with instance weighting we expected the weighting to work well in regression problems, especially with noisy data [7, 34, 97] and our tests with evolutionary instance weighting for regression presented in chapter 9 show that in this case instance weighting is most beneficial.

The results obtained with evolutionary instance selection were better than those obtained with similarity-based instance selection algorithms, although the results obtained with multi-objective evolutionary algorithms are still better. A short comparison will be shown in chapter 9.

8.6 Other Solutions from Literature

Tolvi [98] used genetic algorithms for outlier detection and variable selection in linear regression models, performing both operation simultaneously. The author evaluated the model on two very small datasets (35 instances with 2 features and 21 instances with 3 features).

In [99] an algorithm called Cooperative Coevolutionary Instance Selection (CCIS) was presented. The method used populations evaluated cooperatively. The training set was divided into n equal parts and each part was assigned to a subpopulation. Each individual of a subpopulation encoded a subset of training instances. Every subpopulation was evolved using a genetic algorithm. The second population consisted of combinations of instances sets. The population of combinations kept track of the best combinations of selectors for different subsets of instances, selecting the combinations that are promising for the final global selector selection of the whole dataset.

In [100] Antoneli et al. presented a complex genetic algorithm-based solution. They tackled the instance selection problem with Multi-objective Evolutionary Fuzzy Systems through a coevolutionary approach. In the execution of the learning process, periodically, a single-objective genetic algorithm evolved a population of reduced training sets. The single-objective genetic algorithm aimed to maximize the index, which measured how much the results computed by using, respectively, the reduced training set and the all training set were close to each other - the closer the better.

In [101] a coevolutionary algorithm was presented for instance and feature selection and particular problems were assigned to several populations to handle each one separately. This allowed to employ a divide-and conquer strategy, where each population was optimizing a part of the problem. Then the authors tried to joint the solutions obtained by each population in an attempt to obtain better results as those generated by non-coevolutionary approaches.

In [102] multi-selection of instances was proposed by allowing the selection of instances more than once. If the inner evaluation algorithm in the instance selection is k-NN then for classification tasks an instance can be selected between 1 and k/2 times (more than k/2 does not make sense, as k/2 is enough for correct classification). The advantage of this approach is first reduction of distance matrix calculation time, second reduction of storage, however one additional field must be added to the dataset for the number of each instance repetitions.

In [103] an evolutionary instance selection algorithm that combines three strategies was proposed. The first strategy was to use the optimal genetic algorithm. In this case it was the CHC genetic algorithm, as according to the authors' tests it performed the best from the methods they examined. The second strategy was to incorporate the possibility of selecting each instance more than once (as it was in the paper mentioned above). The third strategy was to use a local value for k that depends on the nearest neighbors of every test instance. So in general their approach was the same as ours to combine several best strategies into one algorithm (and obviously test it they also work best when used together). The difference was however, that they optimized different parameters.

8.7 Conclusions

An instance selection method using genetic algorithms as a search engine was presented. As the experimental results showed, the quality of the solution was usually higher than that of the best similarity based instance selection algorithms, while the computational time was comparable, although a bit longer for small datasets.

We added the caching of the information required by k-NN - the precalculated and sorted distance matrix and the corresponding matrix with instance numbers and outputs. Instance weighting did not cause significant improvement in classification tasks in comparison with binary instance selection. Also setting the balance α between accuracy and compression in some cases did not have any influence on the results, as it was enough to select only a few instances to obtain the best classification accuracy.

However, based on our tests with the methods with different parameters, we noticed, that there are two limitations of all the methods: one is the problem with too long chromosomes, and the other one is the problem of the natural diversity in the dataset, causing possible over-fitting if the optimization is run too long, as was already discussed. Several improvements, which can address these problems and reduce the computational effort are discussed in chapters 10 and 11.

It is very difficult to make the experimental comparison with other evolutionary instance selection methods, as the authors very rarely provide the results in a form that can be compared or the software they used in the experiments, so we were not able to find any results in the literature that could be compared here.

Chapter 9
Multi-Objective Evolutionary Instance Selection

Abstract This chapter focuses on multi-objective evolutionary instance selection in regression and classification tasks. The objectives are: improving prediction results in terms of classification accuracy or $rmse$ and reducing the dataset size. The multi-objective approach allows for obtaining a pool of solutions that create a Pareto front and choosing the desired solution from the front. We analyze how particular properties of the data and parameters of the instance selection process influence the obtained results and suggest the best solutions for given situations.

9.1 Introduction

Instance selection is by its nature a two-objective problem. The first objective is training dataset size minimization and the second one is improvement of the prediction accuracy of the model trained on that data. Thus there are usually multiply solutions to that problem, depending on the coefficient α in Eq. 8.1 and 8.2.

So far to obtain a set of solutions we would have to run the instance selection process many times with different α parameters. However, the multi-objective methods come with help and allow us to run the process only once to obtain a pool of solutions without even defining the α parameter.

In this chapter we present the methods of multi-objective evolutionary instance selection for classification and regression tasks, which we called MISC2 and MISR2 (where 2 stands for the second version of the algorithm). We proposed the first version in our works [104] and [105]. The second version uses improved initialization, new mutation scheme and forms the solution from several subsolutions to improve the results.

First we present the basics of the multi-objective evolutionary instance selection (objectives, the Pareto front, and the instance selection process). Then we will dis-

cuss the optimizations of various parameters. In the next two chapters we will present additional improvements to the process. We will use more space for discussing the multi-objective evolutionary instance selection than the approaches discussed so far, first, because is a more complex issue and second, because so far it allowed us for the best results in terms of accuracy-compression or $rmse$-compression balance.

9.2 Encoding

In multi-objective evolutionary instance selection we used the same encoding of instances into chromosome as in single-objective selection. Each individual in the population represents one training dataset $\mathbf{T}$ and each position of the individual's chromosome represents a single instance; 0 at this position means it is rejected and a positive value - it is selected (1 for binary instance selection and a real number between 0 and 1 for instance weighting). For more details see chapter 8, section 8.2.

9.3 The Objectives

In multi-objective instance selection, we have the same two base objectives as the single-objective selection. However, the difference is that now we do not need any coefficient α to assigned the weights to them as in Eq. 8.1 and 8.1. They are just two separate objectives fo be minimized:

$retention$ and $1 - accuracy$ for classification tasks
and
$retention$ and $rmse$ for regression tasks.

A key advantage of the multi-objective evolutionary optimization is obtaining a pool of solutions situated on the Pareto front (see chapter 7), where each of them is the best for certain $retention$ - $accuracy$ or $retention$ - $rmse$ balance.

Also in the same way, as in single-objective evolutionary-based instance selection, it must be distinguished between the final objective, which is maximization of accuracy or minimization of $rmse$ on the test set (acc_{tst} or $rmse_{tst}$) and the objective used by the instance selection process, which is maximization of accuracy or minimization of $rmse$ on the training set (acc_{trn} or $rmse_{trn}$). The final objective (acc_{tst} or $rmse_{tst}$) cannot be optimized directly, because the test set is not available while selecting the instances.

However, acc_{trn} or $rmse_{trn}$ are not optimized directly, but the NSGA-II algorithm (see chapter 7) optimizes the following values for binary instance selection:

$$\begin{aligned} rmse_{trn}\left(\mathbf{w}\right) &= knn(\mathbf{S}(\mathbf{w}), \mathbf{T}) \\ ret\left(\mathbf{w}\right) &= \tfrac{1}{N}\sum_{i=1}^{N} nred\left(w_i\right) \end{aligned} \tag{9.1}$$

While for real-value instance weighting the following objectives are directly optimized by NSGA-II:

$$\begin{aligned} rmse_{trn}\left(\mathbf{w}\right) &= knn(\mathbf{S}(\mathbf{w}), \mathbf{T}) \\ ret\left(\mathbf{w}\right) &= \beta \cdot \tfrac{1}{N}\sum_{i=1}^{N} w_i + (1-\beta) \cdot \tfrac{1}{N}\sum_{i=1}^{N} nred\left(w_i\right) \end{aligned} \tag{9.2}$$

where $rmse_{trn}(\mathbf{w})$ is obtained with the k-NN algorithm while predicting output of all instances from the original training set $\mathbf{T}$, using the reduced (selected) training set $\mathbf{S}$ given by the weight vector $\mathbf{w}$ (each time without the instance currently being predicted). $ret(\mathbf{w})$ is the sum of instance weights (which in binary selection equals the number of selected instances, while in instance weighting it is a weighted sum of the instance weights w_i and the number of instances with non-zero weights $nred(w_i)$). $nred(w_i)$ returns 1 if the instance is selected and 0 if it is rejected. β is a parameter that balances the sum of the instance weights and the sum of the not rejected instances. The first term, summing the instance weights, is needed to allow crossover and mutation operations to gradually reduce some of the instance weights w_i. For instance weighting $nred(\mathbf{w})$ is calculated as follows:

$$nred\left(\mathbf{w}\right) = \begin{cases} 1 \text{ for } \ w_i > \gamma \\ 0 \text{ for } w_i <= \gamma \end{cases} \tag{9.3}$$

where γ is a rejection threshold. Instances with weights lower than γ get rejected and are not taken into account by the k-NN algorithm, while instances with weights greater than γ are taken into account proportionally to their weights (as well by the inner evaluation as by the final prediction model), as will be discussed in the next section. We set γ to 0.01, however, the exact value is not so crucial, as the algorithm adjusts its behavior by modifying the weights proportionally to γ). Such an approach allows the optimization algorithm to minimize the instance weights and, consequently perform the reduction of instances. When we report the retention in the experimental results (columns $c1$ and $c2$ in the result tables), we take into account the number of all instances with non-zero weights (any non zero weight is counted as 1 while calculating $c1$ and $c2$).

We also attempted to obtain solutions with high prediction quality and low compression that were not present in the main front by using an additional α parameter in the objectives for classification tasks:

$$\begin{cases} obj_0(\mathbf{S}) = 1 - acc(\mathbf{S}) \\ obj_1(\mathbf{S}) = \alpha \cdot (1 - acc(\mathbf{S})) + (1 - \alpha) \cdot retention(\mathbf{S}) \end{cases} \tag{9.4}$$

and for regression tasks:

$$\begin{cases} obj_0(\mathbf{S}) = rmse(\mathbf{S}) \\ obj_1(\mathbf{S}) = \alpha \cdot rmse(\mathbf{S}) + (1 - \alpha) \cdot retention(\mathbf{S}) \end{cases} \tag{9.5}$$

where $\mathbf{S}$ is the selected set, α is a coefficient introduced to adjust the position of the Pareto front by stronger preferences of lower $rmse$ (higher accuracy) or lower retention. If α is set to 0, the objectives of the algorithm work as in a typical Pareto-based algorithm. If α is set to 1 the algorithm works as typical single-objective algorithm. However, these experiments with the α parameter were not very successful and we decided to use multiply Pareto fronts instead (see chapter 10).

9.4 Pareto Front

We present the Pareto front in Fig. 9.1 and 9.7, where each pair of points (orange and green) represents one solution with the percentage of selected instances on the horizontal axis and the corresponding $1 - accuracy$ or $rmse$ on training set (orange) and on test set (green) on vertical axis. Only the points that formed the Pareto front are shown in the figures. Horizontal orange and green lines show the $accuracy$ or $rmse$ without instance selection. In case of classification we place $1 - accuracy$ on the vertical axis in order to make these plots look in the same way as the plots for regression tasks, so that the lower value on the vertical axis is always better.

An example of the obtained Pareto front and the four points of interest are shown in Fig. 9.1. As there is no way to show the whole Pareto fronts for many dataset without extending this chapter far beyond the acceptable length, and moreover no simple way to compare the whole fronts, we will present four most characteristic points of interest $(r0, c0)$, $(r1, c1)$, $(r2, c2)$, $(r_{min}, c(r_{min})$, where $r_{min} = min(r0, r1, r2, r3)$, which we proposed first time in the work [105]:

- **r0** - $rmse$ obtained in 10-fold cross-validation with a given prediction model without instance selection.
- **c0** - $c0$ is always 1, which means there is no compression on the whole dataset, for that reason the column does not occur in any table.
- **r1** - $rmse$ obtained in 10-fold cross-validation with a given algorithm with instance selection corresponding to the point $(c1, r1)$ in the Fig. 9.1; this is the $rmse$ obtained on the test set, using the training set, which started the Pareto front. The figure shows the points on the Pareto front obtained only on one cross-validation fold (one pair of traning-test) in the cross-validation. The values reported in the ta-

bles are averaged over the 10 folds. In orange - the points obtained on the training set and in green the corresponding points obtained on the set set.

- **c1** - retention (1-compression) corresponding to the point $(c1, r1)$ in the Fig. 9.1.
- **r2** - $rmse$ obtained in 10-fold cross-validation with a given algorithm with instance selection corresponding to the point $(c2, r2)$ in the figure; this is the $rmse$ obtained on the test set, using the training set, which was closest to point ($r0_{trn}$, 1-compression=0); the blue line shows this distance. This point was selected as a representative point, because further increasing compression usually leads to sudden increase of $rmse$, making the area to the left of that point practically unusable.
- **c2** - retention (1-compression) corresponding to the point (c2,r2) in the Fig. 9.1.
- **r3, c3** - $rmse$ and compression obtained with the alternative initialization (90% probability of each instance being included in the initial population) and objective criteria - only $rmse$ was minimized; it was aimed to obtain $rmse$ lower than r1 and r0 (without compression). However, sometimes it happened that $r3$ was equal $r0$ or to $r1$, what meant that no further decrease of $rmse$ below $r0$ or $r1$ was possible to obtain.

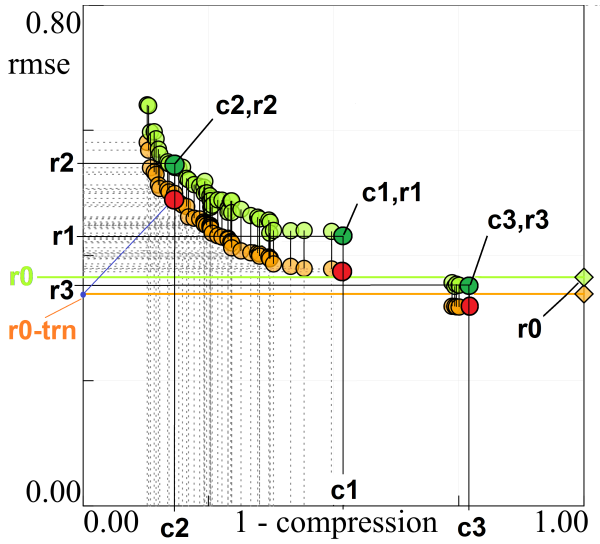


Fig. 9.1. Sample results in one fold of the cross-validation. They symbols used in the figure are described in the listing in Section 9.3.

9.5 Instance Selection Process

The diagram of the process is presented in Fig. 13.1 and 13.2. For the NSGA-II based optimization we use p=96 individuals (96 is a multiply of the number of CPU cores in our server, so it is more optimal than e.g. 100 due to the process parallelization). The number of epochs (iterations) in the evaluation must be carefully chosen, because if too many epochs are used, an over-fitting occurs and the results on the test set begin to drop although the results on the training set are still improving. Based on the experiments, we decided to use the following number of epochs e rounded to the nearest integer:

$$e = e0 \cdot log(N) \tag{9.6}$$

where N is the number of instances in the original training set **T**, $e0$=8 for binary instance selection and $e0$=24 for real value instance weighting. For example, that gives 20 epoch for 300 instances and 34 epochs for 30,000 instances for binary instance selection.

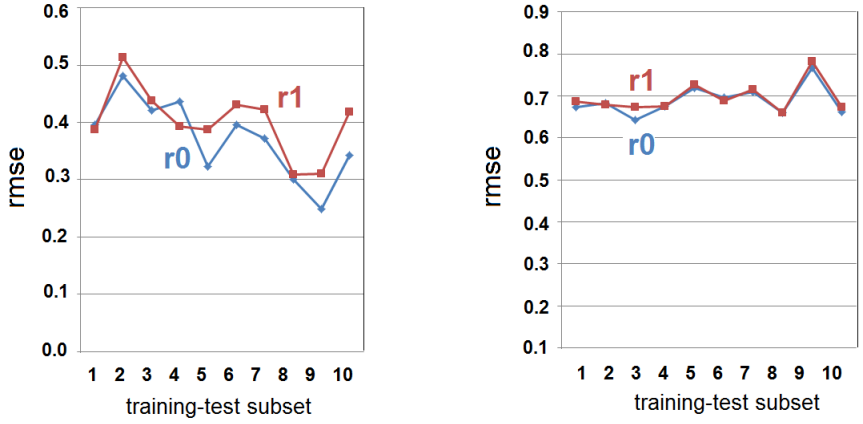


Fig. 9.2. Horizontal axis: 10 pairs of training-test subsets in 10-fold cross-validation. Vertical axis: $rmse$ obtained on each test subset. The blue points (r0) represent the $rmse$ for the 10 test subsets before instance selection and the brown points (r1) after the selection: left on a small dataset (autoMPG8), right a medium-size dataset (abalone).

The random variability of $rmse$ introduced by the instance selection is really very low; the $rmse$ obtained on each test subset within the cross-validation after the compression is highly correlated with that one before the compression. Thus the variance

obtained within cross-validation was mainly caused by the diversity of the datasets and not by the stochastic character of the evolutionary optimization. This is illustrated in Fig. 9.2, where the blue points represent the $rmse$ for the 10 test subsets before instance selection and the brown points after the selection. In Fig. 9.2 it is shown for a small a medium-size dataset. For large datasets the correlation is even higher.

The rationale behind choosing k-NN as the inner evaluation algorithm is the speed of this approach, as described in the previous chapter.

In the following four sections particular steps of the optimization are presented and finally the conclusions from the optimization are combined together. In each of the sections first description of the problem is provided and then experimental results are presented. The experiments are short in the first two optimizations, because the choices of the parameters are clear and much more detailed in the last two optimizations, where the parameters can be chosen depending on data properties and user preferences and thus require more discussion.

9.6 Choice of the Multi-Objective Genetic Algorithm and its Parameters

9.6.1 Problem Description

The first point to decide is which multi-objective evolutionary algorithm should be used. We conducted a lot of tests to choose the best method. There is no place here to present all of them, thus we will only show some examples to illustrate the problem and final conclusions. In the tested methods we included not only the multi-objective solutions, but also considered the possibility that if we run several times single-objective evolutionary algorithms to obtain several solutions on the Pareto front, the results will be so much better that the only advantage of multi-objective approaches will be time saving at the cost of the solution quality. However, that turned out not to be true.

9.6.2 Experimental Evaluation and Discussion

We have evaluated several evolutionary algorithms and made a literature search to find the comparisons in other studies. There was the following conclusion: we should choose the NSGA-II algorithm [94] and adjust it to our purposes. Despite newer solutions, it often produces the best results. Recently, a new version of this algorithm has been developed (NSGA-III [95]), which is dedicated to problems with more than two objectives. As both algorithms were developed by Deb, there is no reason not to

believe him, when he suggests to use NSGA-II for problems with two objectives and NSGA-III for problems with more than two objectives. For this reason, we did not considered and not tested NSGA-III. Because we decided to base on NSGA-II, we described it in chapter 7, section 7.10.

Here we present a short comparison of three selected methods:

1. single-objective genetic algorithm as used in chapter 8
2. NSGA-II
3. SEEA

and evaluation of the following parameters:

1. population size ($P = \{24, 48, 96, 192\}$)
2. The α parameter in fitness function in Eq. 9.5 ($\alpha = \{0, 0.5, 1\}$); this value refers to the second objective (α=0 means the second objective is retention, α=0.5 means the second objective is $0.5 \cdot retention + 0.5 \cdot rmse$, α=1 means both objectives are $rmse$)

Some of the obtained results are presented in Table 9.1 and 9.2. The examples of the results for selected problems are shown in Fig. 9.3 and 9.4. From that analysis we draw the following conclusions:

- The NSGA-II algorithm showed the best performance. It allowed for obtaining the lowest $rmse$ and the widest variation in the obtained results in terms of compression (with α=0 in Eq. 9.5). In even allowed obtaining a lower front than several runs of a single-objective genetic algorithm.
- The lowest $rmse$ was obtained for $\alpha = 1$, however in that case it was not possible to obtain a wide spectrum of solutions, and the compression level was low in comparison with other results.
- The population of about 60-100 individuals seems to be optimal. Above this the $rmse$ value and the range of differentiation in terms of compression does not improve significantly anymore. Below that value, the convergence can be in some cases a little faster, that however limits the population diversity and the process becomes less stable, resulting in not finding the best results sometimes. We decided to use the population size P=96 individuals, because there were 24 and 48 CPU cores in our servers and that allowed to scale the calculations well in parallel.

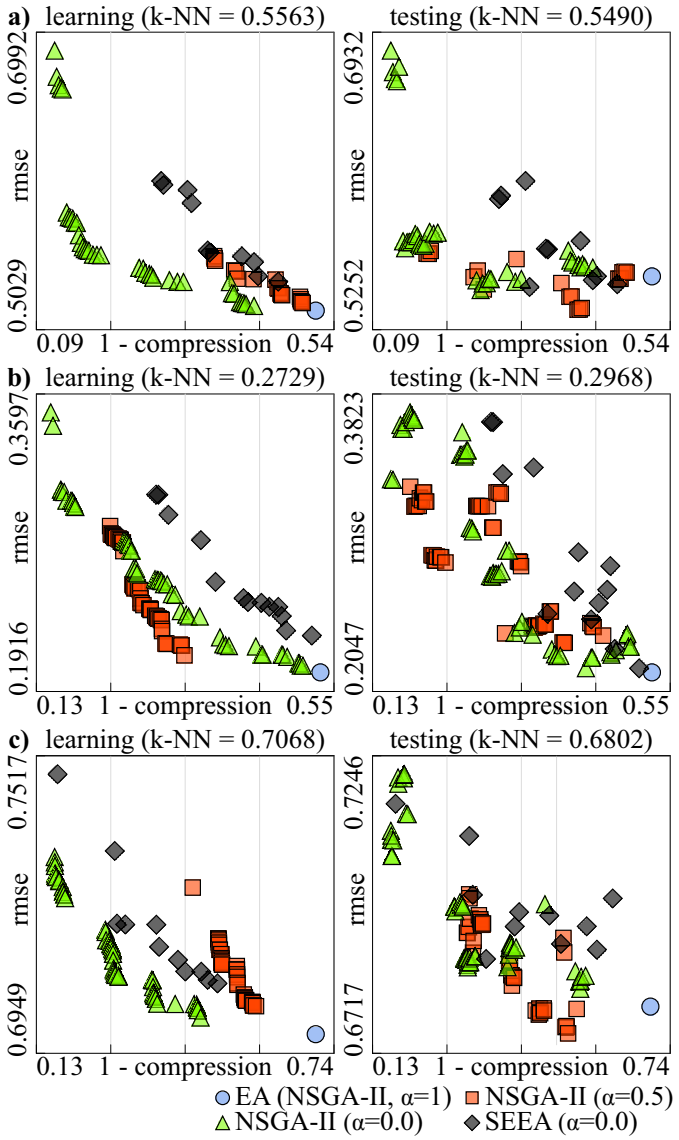


Fig. 9.3. Example of Pareto fronts obtained with different evolutionary algorithms using k-NN with optimal k as inner evaluator and final predictor for three datasets: a) ele-1, b) laser, c) abalone.

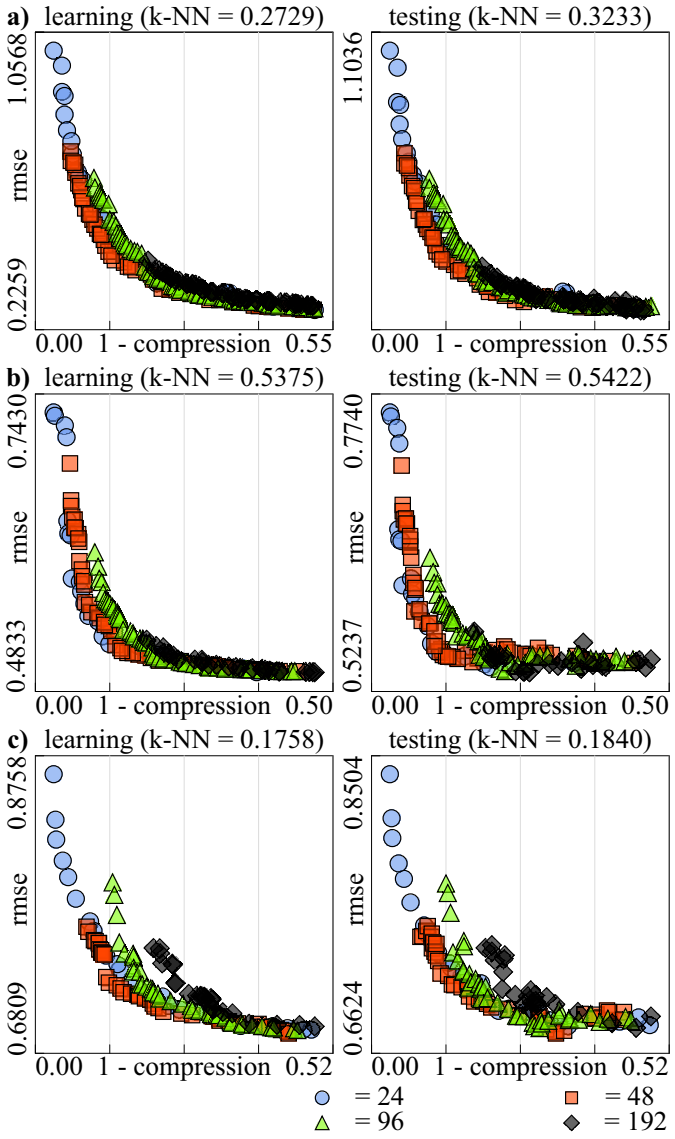


Fig. 9.4. Example of Pareto fronts obtained with NSGA-II with different population sizes P = 24, 28, 96 and 192, using k-NN with optimal k as inner evaluator and final predictor for three datasets: a) ele-1, b) laser, c) abalone.

Table 9.1. Average results for all tested datasets for different evolutionary algorithms.

algorithm	r1	c1	r2	c2
EA (NSGA-II, $\alpha = 1.0$)	0.536	0.472	0.536	0.472
NSGA-II ($\alpha = 0.5$)	0.544	0.422	0.572	0.223
NSGA-II ($\alpha = 0.0$)	0.542	0.442	0.580	0.174
SEEA ($\alpha = 0.0$)	0.551	0.430	0.569	0.330

Table 9.2. Average results for all tested datasets for different population size P.

P	iterations	r1	c1	r2	c2
24	160	0.552	0.385	0.593	0.201
48	80	0.547	0.417	0.585	0.162
96	40	0.539	0.435	0.580	0.168
192	20	0.543	0.422	0.592	0.220

9.7 Assessment of Population Initialization Methods

9.7.1 Problem Description

Typical initialization assigns randomly generated values as parameters of the individuals. For this purpose different methods can be used (e.g. Pseudo-Random Generators [106], Quasi-Random Generators [107], Population dependent methods [108]). The main goals of such methods is the optimal and even coverage of the search space, which can increase convergence speed of the optimization and decrease standard deviation of the obtained results.

In addition, methods based on value transformations [109], a priori knowledge about the problem [110] or clustering [111] can be also used. It is worth mentioning that, according to some authors, initialization should be selected to a particular issue, and even to a specified simulation problem [112]. In our study different types of the initialization, including the proposed initialization adapted to the complexity aspects of instance selection, were tested.

According to [108], one of the best initialization method that relies on population is Adaptive Randomness (AR). In AR only this candidate for which the smallest distances to the rest of the population is larger than for all other candidates is added to the population. The advantage of this method is that the created candidates do not have to be evaluated, only the distance to the already assigned individuals is calculated. Population-based methods do not depend on random number generators and methods of transforming numbers. Due to that they can be combined with other types of initialization methods [113]. We tested such combinations with the idea to produce different initial values (e.g. initialize more smaller values and examine the effect of such ini-

tialization on the resulting compression). The tested initialization methods are shown in Table 9.4.

Additionally, we used new methods developed by Krystian Łapa, which aim at differentiation of initialized values (to achieve different degrees of compression in instance selection):

- Power transformation. In this method the randomly generated number rnd is raised to a certain power v as follows:

$$w = rnd^v \tag{9.7}$$

  where higher v results in lower values obtained after transformation (because rnd<1).
- Fill transformation. This binary transformation method uses probability f that determines balance between 0 and 1 in the initialization:

$$x = \begin{cases} 0 \text{ for } rnd < f \\ 1 \text{ for } \quad \text{else} \end{cases} \tag{9.8}$$

- Spread initialization. The idea behind this method is to differentiate the individuals in population in terms of the different probabilities of occurrence of small and large values:

$$x = \begin{cases} 0 \quad \text{for } rnd < c \\ rnd \text{ for } \quad \text{else} \end{cases} \tag{9.9}$$

  where $c = 0.1 + h * n/S$ is a parameter dependent on the individual in the population, n stands for index of the individual in the population, h is the method parameter (see Table 9.3).

The numbers generated by initialization methods are rounded to the closest acceptable values defined by the $numLevel$ parameter (see Section 9.9).

Initially we thought about using a similarity-based instance selection algorithm, as CNN or DROP5, to initialize the population with the results of that algorithm and then only make some random perturbations around these values. But this turn out to be impractical. There are two reasons for that. The first one is that for big datasets DROP5 is slower than our method, so this does not make sense. CNN with some modification to improve the k-NN calculations can be faster. However, using that kind of initialization is not optimal, as this significantly narrows the initial pool of solutions and thus first the genetic algorithm has to produce the more evenly spread solutions, what together does not bring any time gain and increases the complexity of that approach. Thus it is a better option to use some the initialization methods from Table 9.3, adjusted to the expected Pareto front location (see chapter 10).

9.7.2 Experimental Evaluation and Discussion

The population initialization methods and the obtained results are presented in Table 9.3 and in Fig. 9.5. The results can be summed up as follows:

- Initialization methods, especially those diversified in terms of fill value level of the initial population, have a significant influence on the obtained results.
- Most methods (except 3, 4, 7, 11) do not allow a high degree of compression. These four methods seem to be the most interesting in combination with further use of the NSGA-II algorithm.
- Very good accuracy (low $rmse$) can be obtained using methods 9 and 10, unfortunately in this case the results have the lowest degree of compression.

Table 9.3. Average results for all tested datasets for different initialization methods (lower values are better). See section 9.4 for explanation of the symbols *r1, c1, r2, c2*.

init	transformation	population	r1	c1	r2	c2
1	adjusted to acceptable values	simple	0.539	0.481	0.550	0.423
2	none	simple	0.541	0.457	0.565	0.281
3	power ($p = 2$)	AR ($R = 10$)	0.542	0.434	0.613	0.191
4	power ($p = 4$)	AR ($R = 10$)	0.535	0.447	0.597	0.130
5	power ($p = 8$)	AR ($R = 10$)	0.540	0.477	0.554	0.411
6	fill ($f = 0.500$)	AR ($R = 10$)	0.539	0.437	0.565	0.260
7	fill ($f = 0.250$)	AR ($R = 10$)	0.539	0.422	0.583	0.175
8	fill ($f = 0.125$)	AR ($R = 10$)	0.541	0.540	0.555	0.420
9	fill ($f = 0.750$)	AR ($R = 10$)	0.533	0.603	0.549	0.433
10	fill ($f = 0.875$)	AR ($R = 10$)	0.535	0.607	0.546	0.434
11	none	spread ($h = 0.2$)	0.534	0.565	0.568	0.204
12	none	spread ($h = 0.8$)	0.543	0.429	0.569	0.261

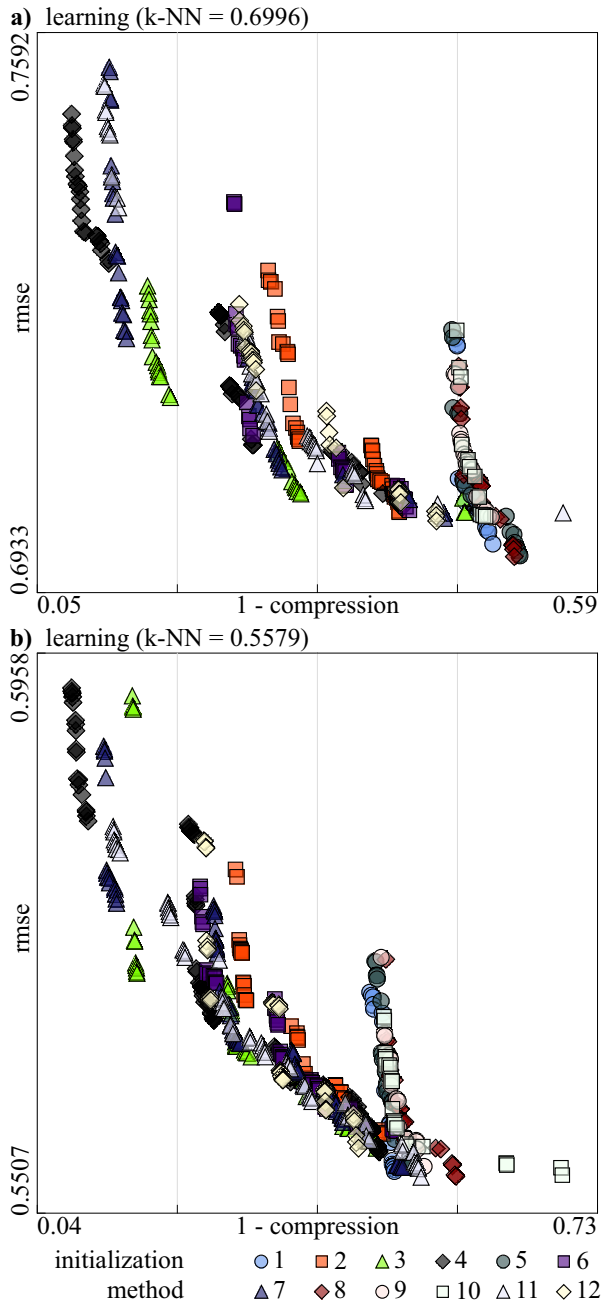


Fig. 9.5. Example of Pareto fronts obtained with NSGA-II and different initialization methods using k-NN with optimal k as inner evaluator and final predictor for three datasets: a) ele-1, b) laser, c) abalone. See Table 9.3 for the number explanation.

9.8 Tuning k-NN Parameters

9.8.1 Problem Description

In the first and second step we were able to choose the most optimal solutions for our purposes, that were universal for most of the cases. Here, we cannot specify one universal set of parameters, which is best for all the evaluated models. The optimal parameters vary depending on the dataset and the final predictive model. For that reason we provide here more detailed test results, which are placed in the next chapter in order to compare them also with the improved versions, described in the next chapter.

9.8.2 Experimental Evaluation and Discussion

Most of the papers on instance selection (however not all fortunately) show only how instance selection influence the prediction in classification or sometimes regression tasks, when the predictive model is 1-NN algorithm. Indeed 1-NN is rarely the optimal prediction model and it is much easier to improve the results of a poor model (1-NN) than of a good one. Examples of better models are k-NN with optimal k, an MLP neural network and other models (of course these models do not perform better than 1-NN in each case, but on average they definitely do). So in this way the authors can easily show how much their methods improved the results, but they cannot show how universal the method is in improving the results with other predictive models.

In this chapter we have chosen three predictive models: 1-NN, k-NN with optimal k, and an MLP neural network. The three models were chosen purposely, as each of them represents one group of predictive models with different properties, thus by evaluating the results on these models we can asses the performance of the methods with other learning models.

1-NN is very sensitive to instance selection, as it predicts the output value of the instance of interest to be the same as its nearest neighbor value. Thus, when we change the nearest neighbor by removing the current one, the prediction will significantly change.

k-NN is less sensitive to instance selection, the higher k the less. That is, because k-NN averages the outputs of k instances. If we remove one of them, then another instance will take its place. This will change the outcome, but definitely less strongly than the outcome of 1-NN. Even, if we remove more of the current neighbors and they will get replaced by the next nearest remaining instances, then usually some of them will have lower and some higher value of the output (or in case of classification some will change the class from A to B and other from B to A), so the changes will significantly cancel out one another.

Finally an MLP neural network belongs to the third group of learning models. An MLP network displays several properties, that were absent in k-NN. The first one is the possibility of any shapes of the decision boundary in classification or any shape of the target function in regression. The next one is that a very broad neighborhood is used to form the decision boundaries or the target function. The third property is that an MLP neural network performs automatic feature weighting during the learning process, done by setting the weights between neurons to appropriate values (see chapters 14 and 17). Although, we can learn the k-NN model using the weighted features, e.g. using correlation in regression problems or another feature filter, its prediction will never be exactly the same.

The results of this section can be summarized as follows:

- A similar compression was achieved for 1-NN and k-NN as the inner evaluators. However, the biggest improvement in terms of $rmse$ was observed for 1-NN, where the average $rmse$ reduction was 4.0% for retention rate 62.7% and 9.0% for retention rate 74.0%. 1-NN was the algorithm that produced the highest $rmse$ on the original uncompressed data and thus there was the biggest room for improvement in this case. In case of optimal k k-NN the algorithm performed better on the original uncompressed data in point $c1$ with retention rate = 49.7%. However, at the point $c3$ with retention rate 84.3% we were able to reduce $rmse$ by 1.6%.
- The MLP neural network was less sensitive to the instance selection in terms of the $rmse$ and the most obvious benefit of instance selection in this case was the shortening of the network learning process and thus giving the chance to try many different network configurations in a limited time. Although the $rmse$ decreased by 0.8% at $c1$ and by 2.5% at c3 - the first change was statistically insignificant according to as well t-test and Wilcoxon test. The MLP obtained definitely the best results for these problems on the uncompressed data and proved to be a better classifier than any k-NN algorithm. Thus, it was quite difficult to improve the results of MLP prediction but also difficult to make the prediction worse by too strong instance selection. At the point $c2$ (with the strongest compression) the increase of the $rmse$ was about 3 times lower than for 1-NN and k-NN. However, as we have already mentioned, the results could be improved by using the MLP network also as the evaluation algorithm on the training set during the evolutionary optimization, but as it causes very high growth of computational complexity, the solution is in most cases not advised. However, second best evaluation algorithm to work with an MLP neural network predictor was the optimal k k-NN.
- It is interesting that the optimal k was frequently different before and after instance selection. After instance selection it tended to converge more or less to the values of 5, 6 or 7. That is, if before selection the optimal k was 1 then it could be 3 after selection, if it was 3 before it was about 5 after, if it was 11 before, it was about 7 after. It can be explained, as after the selection fewer instances remained in the dataset, so the distances between them were bigger and frequently the previous closest neighbor of the examined instance no longer existed and thus it had to be

replaced by some further, but still existing instances. On the other hand a high value of optimal k is characteristic for noisy datasets. As instance selection removes noise and outliers, thus no longer so many neighbors are required to mask the detrimental effect of the outliers.

The tables with results are presented in chapter 10 together with the results obtained with the additional enhancements used in MEISR version 2.

9.9 Evaluating Instance Weighting Scheme

9.9.1 Problem Description

In a typical instance selection algorithm a binary vector of parameters stores the information about the selected instances. However, to enhance the performance of instance selection algorithms, a weight for each instance can be additionally assigned. This can be really important especially in the regression problems [96, 114, 7]. This topic was already discussed in the previous chapter. Here, we add more possibilities to the binary and real-value weights.

In this approach each individual is encoded as a weight vector $\mathbf{w} = \{w_1, ..., w_N\}$, where N is the size of the training set T. What is new, however, is that the weight vector $\mathbf{w}$ can only take a specific values assigned from the set $\{0, \sigma, 2\sigma, 3\sigma, ..., 1\}$, where the σ is determined by the parameter $numLevels$ and it is calculated as follows: $\sigma = 1/\left(numLevels - 1\right)$. In case of $numLevels$=2 the weight vector $\mathbf{w}$ can have only two values: 0 and 1. In case of $numLevels$=5 the weight vector $\mathbf{w}$ can have five following values: 0.00, 0.25, 0.50, 0.75, 1.00, etc. If $numLevels$=0 the weight vector $\mathbf{w}$ can be assigned any real number between 0 and 1. The initial values of vectors depend on the initialization method. Due to the proposed encoding, the initialization, crossover and mutation operators used for a modification of the individuals by genetic algorithm have to bee also adjusted (see Table 9.4).

9.9.2 Experimental Evaluation and Discussion

The following values of the $numLevels$ parameter have been evaluated:

1. $numLevels = 0$ (real values).
2. $numLevels = 2$ (binary values).
3. $numLevels = 5$ (0.00, 0.25, 0.50, 0.75, 1.00 values).
4. $numLevels = 11$ (0.0, 0.1, ..., 0.9, 1.0 values).

Table 9.4. Proposed crossover and mutation operators, rnd is a randomly generated real number from the range $\langle 0, 1\rangle$, $RND(n1, n1)$ is a randomly generated integer number from the set $\{n1, ..., n2\}$, m_{range} is the mutation range parameter (set experimentally to 0.2), each gene is mutated with probability m_{prob}=0.2, w^1 and w^2 denotes the values of the parents' genes.

operator	$numLevels$	modification
crossover	0	$x = \begin{cases} w^1 + rnd \cdot (w^2 - w^1) & \text{for } rnd < 0.2 \\ w^1 & \text{for } rnd > 0.6 \\ w^2 & \text{for else} \end{cases}$
crossover	> 0	$w = \begin{cases} w^1 \text{ for } rnd < 0.5 \\ w^2 \text{ for else} \end{cases}$
mutation	0	$w = w + (rnd - 0.5) \cdot m_{range}$
mutation	> 0	$w = \sigma \cdot RND\,(0, numLevels - 1)$

The obtained results are presented in Table 9.5. The examples of the results for selected datasets are shown in Fig. 9.6. The results can be summed up as follows:

- The lowest $rmse$ was obtained for $numLevels$=5, but the difference was insignificant and the compression was far worse in comparison to $numLevels$=2.
- Adding more possible values that instance weights can take, makes instance selection process longer (more iterations are required).
- The results for $numLevels$=2 are in most cases the best in terms of $rmse$ - compression balance.
- The results for $numLevels$=0 take the broadest range of compression values. The two areas where $numLevels$=0 can provide better results than binary instance selection are the regions of very high compression where it is usually able to achieve lower $rmse$ and noisy datasets that require high k value in k-NN.

Table 9.5. Average results for the examined datasets for different values of $numLevels$ parameter.

$numLevels$	r1	c1	r2	c2
0 (real values)	0.551	0.749	0.618	0.243
2 (binary values)	0.550	0.427	0.580	0.201
5 (0.00, 0.25, ...)	0.543	0.655	0.572	0.402
11 (0.1, 0.2, ...)	0.545	0.763	0.572	0.539

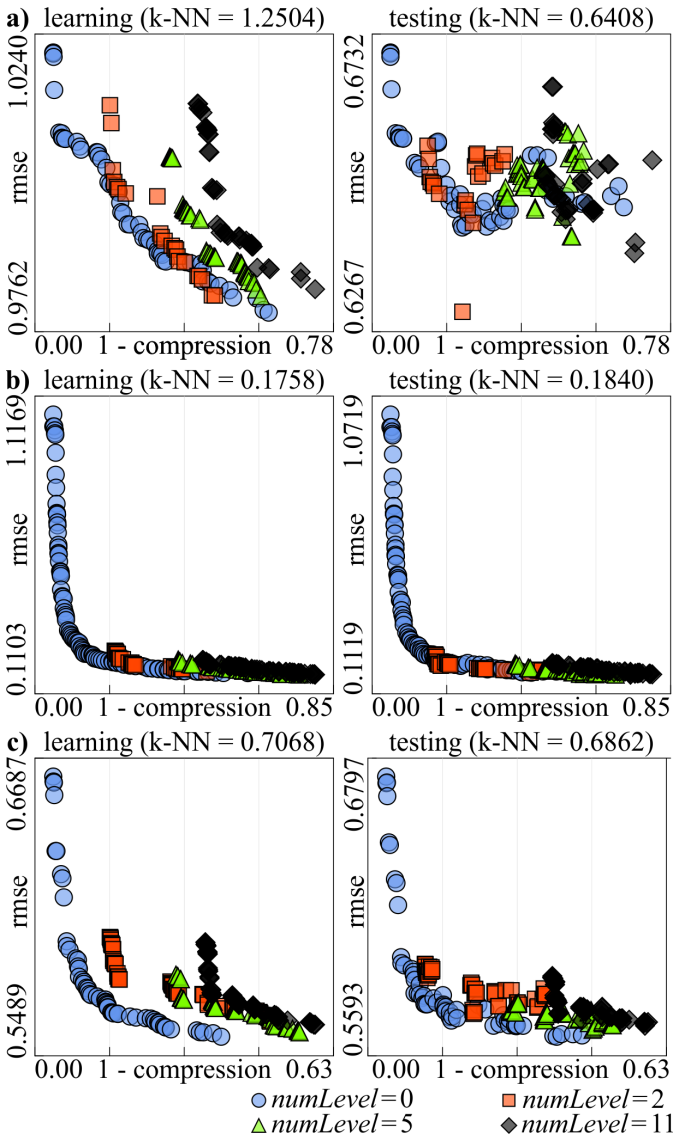


Fig. 9.6. Example results for different values of $numLevels$ parameters for: a) forest fires dataset, b) wankara dataset, c) delta ailerons dataset. Results values for k-NN method with optimal k are also included.

Table 9.6. Experimental results. Inner evaluation model: optimal k k-NN with real-value instance weighting. Prediction model: optimal k k-NN with real-value instance weighting, k=3 was used if optimal k was less than 3.

dataset	rr1	c1	rr2	c2
machineCPU	1.038	0.385	1.086	0.216
baseball	1.029	0.591	1.244	0.110
dee	1.015	0.403	1.182	0.137
autoMPG8	1.108	0.499	1.320	0.160
autoMPG6	1.082	0.401	1.248	0.156
ele-1	0.963	0.489	0.985	0.178
forestFires	0.844	0.456	0.847	0.126
stock	1.338	0.689	1.729	0.371
steel	1.110	0.665	1.333	0.165
laser	1.193	0.489	1.330	0.128
concrete	1.138	0.593	1.303	0.129
treasury	1.503	0.531	2.015	0.159
mortgage	1.554	0.682	2.246	0.263
friedman	1.132	0.668	1.437	0.176
wizmir	1.081	0.491	1.339	0.117
wankara	1.096	0.643	1.635	0.127
plastic	0.965	0.504	1.112	0.184
quake	0.976	0.374	0.979	0.117
anacalt	1.661	0.510	1.904	0.226
abalone	1.028	0.435	1.059	0.136
delta-ail	1.011	0.477	1.035	0.170
puma32h	1.008	0.619	1.035	0.135
compactiv	1.077	0.441	1.388	0.097
delta-elv	1.000	0.546	1.027	0.123
tic	0.970	0.496	0.977	0.121
ailerons	1.015	0.421	1.069	0.128
pole	1.249	0.583	1.319	0.108
elevators	0.990	0.662	1.110	0.081
california	1.008	0.532	1.053	0.112
house	1.046	0.671	1.083	0.236
mv	1.280	0.540	1.328	0.420
average	1.113	0.532	1.282	0.165

As can be seen from Fig. 9.7 using instance weighing we obtain a Pareto front that is more extended and determined by more points, even for small datasets, as autoMPG8.

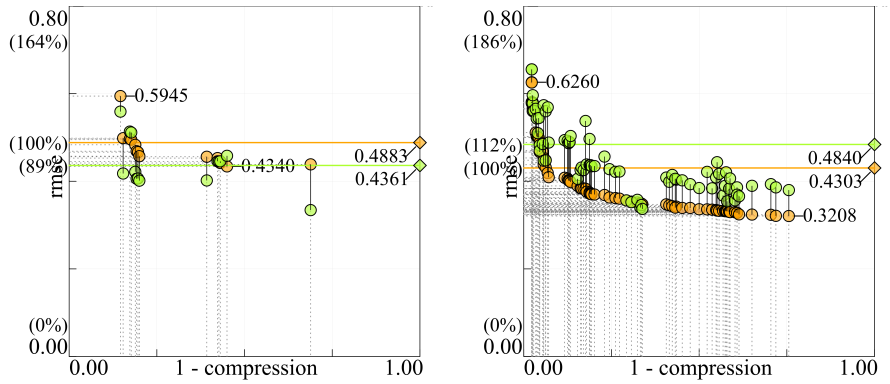


Fig. 9.7. Sample Pareto fronts obtained for the autoMPG8 dataset. Left: binary instance selection. Right: real-value instance weighting. Orange points: training set, green points: test set. Horizontal orange and green lines: $rmse$ without instance selection.

9.10 Comparison with Other Methods

Not counting our own papers, we were able to find in the literature only one paper, which considered instance selection for regression problems with the results presented on several datasets, reporting the obtained compression and $rmse$ or coefficient of determination R^2. So this is another proof that there are not much reliable results on the research in instance selection in regression tasks and that it was a good idea to undertake this research topic.

The paper "Instance selection for regression: Adapting DROP" [30] presented results only for a single point and used 8-NN for each dataset. We obtained from the authors detailed experimental results for most of the datasets from the Keel Repository. So we conducted the experiments with our approach using 8-NN for a precise comparison with that method and measured the output additionally in coefficient of determination R^2, because that measure was used in the paper. There were four methods in this paper and clearly one of them (DROP3-RT) was better than the others, so we included only DROP3-RT in the comparison.

Obviously we could perform a good comparison with our results obtained from other methods. Thus we selected the best four of those methods for comparison. These methods were based on instance selection ensembles, as described in chapter 5.

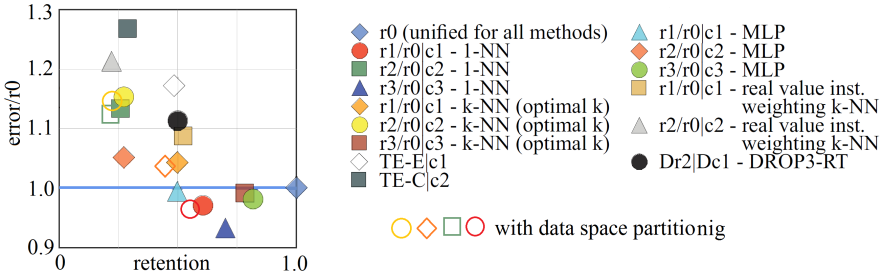


Fig. 9.8. Average results over 30 regression datasets from Keel Repository obtained with various instance selection algorithms. The data partitioning is discussed in the next chapter.

Table 9.7. Comparison of average values over 30 datasets for 5 instance selection methods: the best four methods from the work [38]: Threshold-Ensemble-CNN, Threshold-Ensemble-ENN, Discretization-Ensemble-CNN, Discretization-Ensemble-ENN and the multi-objective evolutionary instance selection. The values in the table are the relative $rmse$ ($rmse$ with compression divided by $rmse$ without compression: $r1/r0$) for retention rates $c1 = 0.5$ and $c2 = 0.25$ (compression = 50% and 75%) obtained in 10-fold cross-validation.

	c1=0.50					c2=0.25				
	TE-C	TE-E	DE-C	DE-E	MEISR1	TE-C	TE-E	DE-C	DE-E	MEISR1
everage	1.201	1.148	1.253	1.153	1.068	1.354	1.537	1.420	1.484	1.227
times best	1	1	0	3	19	1	0	0	1	22

In Tables 9.7 and 9.8 always the lower values are better. The DROP3-RT algorithm produced only a single solution. In order to compare the Pareto front produced by our method with this point, we decided to use always the point with the fifth weakest compression from the Pareto front, as it had much lower retention rate and only very slightly higher $rmse$ than the first point. Only in the case, where the $rmse$ at this point was higher than the $rmse$ of DROP3-RT we switched to the first point (c1), but only if this did not increase retention rate over the one of DROP3-RT.

The MEIRS1 evolutionary optimization allowed for obtaining significantly better results than all the other methods. Only for the largest datasets the $rmse$ obtained with DROP3-RT was in several cases lower, but the compression of MEIRS1 was stronger. And exactly this experiment allowed us to notice that there may be some problem if the chromosome is too long. Later we added the solutions described in the next chapter, which improved the results for big datasets in the MEISR2 method. Detailed experimental results are provided in the next chapter.

Table 9.8. Comparison in 10-fold cross-validation with DROP3-RT for regression datasets for 8-NN. The values in the table: D_{1-R^2}: relative $1-R^2$ (ratio of $1-R^2$, where R^2 is the correlation between the predicted and actual output, with instance selection to $1-R^2$ without instance selection with DROP3-RT algorithm, M_{1-R^2}: relative R^2 with Multi-objective Evolutionary Instance Selection for Regression (MISR1), Mr2/Dr2: ratio of Mr2 to Dr2, Dc1: retention rate (1-compression) with DROP-RT algorithm, Mc1: retention rate (1-compression) with MISR1, Mc1/Dc1r: ratio of Ec1 to Dc1. (source: our work [105])

dataset	D_{1-R^2}	M_{1-R^2}	M_{1-R^2}/D_{1-R^2}	Dc1	Mc1	Mc1/Dc1
machineCPU	1.541	1.263	0.819	0.495	0.374	0.756
baseball	1.160	1.134	0.978	0.460	0.395	0.859
dee	1.166	0.991	0.850	0.511	0.352	0.689
autoMPG8	1.210	1.118	0.924	0.491	0.421	0.857
autoMPG6	1.168	1.137	0.974	0.511	0.373	0.729
ele-1	1.023	0.892	0.873	0.486	0.353	0.725
stock	1.685	1.421	0.843	0.572	0.483	0.843
laser	1.436	1.079	0.751	0.605	0.455	0.752
concrete	1.351	1.235	0.914	0.502	0.465	0.926
treasury	1.346	1.500	1.114	0.620	0.439	0.707
mortgage	1.474	1.600	1.086	0.670	0.480	0.717
friedman	1.076	1.186	1.102	0.538	0.437	0.812
wizmir	1.329	1.140	0.858	0.510	0.410	0.803
wankara	1.355	1.103	0.814	0.521	0.444	0.853
plastic	0.964	0.833	0.864	0.419	0.390	0.930
quake	1.058	1.044	0.986	0.420	0.387	0.922
anacalt	1.551	1.172	0.755	0.483	0.422	0.874
abalone	1.053	1.034	0.982	0.416	0.389	0.935
delta-ail	1.039	1.001	0.963	0.433	0.364	0.841
puma32h	1.032	1.062	1.028	0.381	0.375	0.986
compactiv	1.557	1.066	0.685	0.449	0.391	0.870
delta-elv	1.016	1.011	0.995	0.432	0.376	0.870
ailerons	1.039	1.134	1.092	0.425	0.264	0.620
pole	1.425	1.696	1.190	0.245	0.244	0.996
elevators	1.137	1.135	0.998	0.438	0.358	0.816
california	1.048	1.105	1.055	0.475	0.402	0.845
house	1.084	1.041	0.969	0.430	0.318	0.741
mv	1.061	1.297	1.222	0.531	0.392	0.739
average	1.228	1.161	0.953	0.481	0.391	0.822
times best	21	7		0	28	
t-test p			0.2105			0.0000
Wilcoxon p			0.0657			0.0000

9.11 Classification Problems

The same methodology can be used for classification datasets. Fig. 9.9 shows a sample Pareto front obtained for the classification problem for LEd7 dataset. The only difference is that the first criterion we minimize is $(1 - accuracy)$ instead of $rmse$. We compared experimentally the results of this approach with an ensemble of DROP5 methods. As DROP5 is one of the best similarity-based instance selection methods and ensemble methods performed better than single methods, we wanted to compare with the best alternative approach, which is able to generate Pareto fronts. In both cases we used k-NN with optimal k as the final predictor.

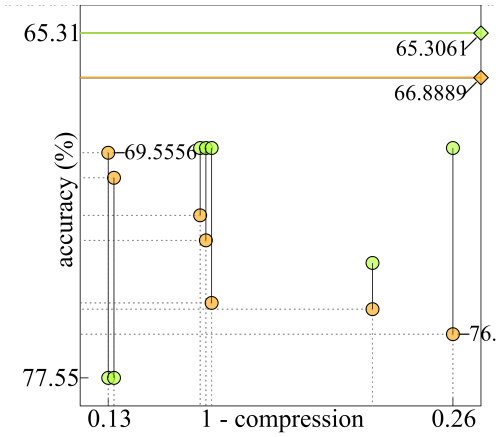


Fig. 9.9. Sample Pareto front for the classification Led7 dataset obtained on a single training-test set pair within a 10-fold cross-validation. The average over the whole cross-validation gives much more regular fronts.

As for regression datasets, also for classification tasks in most cases better results were obtained with the multi-objective evolutionary methods (Fig. 9.10-left). However, similarly, as for regression datasets, there were a few exceptions: where the datasets were big and the classification accuracy obtained without instance selection was very high, than the DROP5 ensembles tends to be better (Fig. 9.10-right).

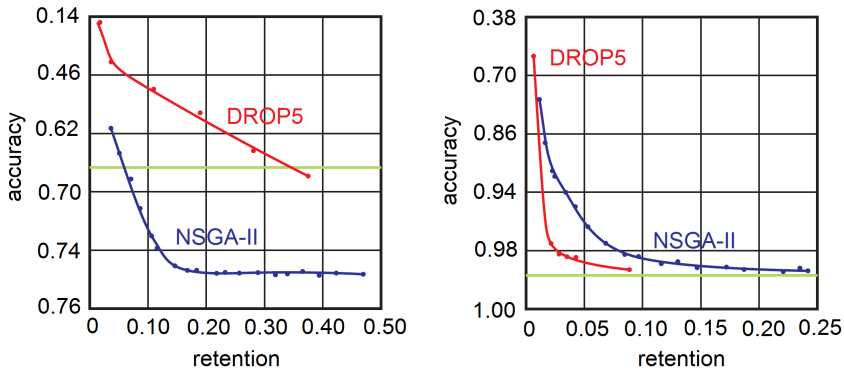


Fig. 9.10. Sample Pareto fronts (average over 10-fold cross-validation) on the test set for the classification datasets: Left: Led7digit - this kind of results we obtained for most datasets. Right: Penbased - only in a few cases the DROP5 ensembles were better. Horizontal green lines show the baseline (accuracy without instance selection). Note: vertical line is in logarithmic scale.

9.12 Other Solutions from Literature

We found only two works describing the application of multi-objective evolutionary algorithms to instance selection, both to classification problems and both dated for late 2017.

In [115] the MOEA/D algorithm was used in a coevolutionary approach integrating instance selection and generating the hyper-parameters for training an SVM just in a similar idea as in [101]. The two criteria used in that optimization were the reduction of the training set and the performance when such a reduction is used with a given set of an SVM's hyper-parameters. The average results over some classification datasets were provided.

In [116] the authors considered also the over-fitting problem. At each iteration of the genetic algorithm the training and validation partitions were updated in order to prevent the prototypes from over-fitting a single validation dataset. Each time the partitions were updated, all of the solutions in the Pareto set were re-evaluated. However, only the accuracy and reduction for 1-NN averaged over several classification problems was reported, similarly, as in the previous work, so it was not possible to compare the results to other methods. Nevertheless, that is an interesting approach. We use instead an early stopping of the optimization or a typical validation set.

9.13 Conclusions

We proposed a methodology that allows to choose the optimal set of algorithms and parameters depending on the dataset properties and on the defined priorities. The following detailed conclusions that show the advantages of the method can be drawn from this chapter:

- The obtained results very in general very good, definitely better than the results of the other compared methods.
- A significant advantage of the methodology is that we can obtain the entire Pareto front of solutions. The user can choose one of them depending on his preferences of the accuracy-compression values.
- k-NN is very well suited as the inner evaluation algorithm, because of its speed - the distance matrix has to be calculated and sorted only once and then the prediction is immediate.
- Proper initialization of the population plays important role, always in accelerating the instance selection process and in some cases also in allowing to find the best solution.

There are however still some areas for improvement, which are listed below and which will be addressed in chapter 10 and 11:

- The first useful improvement will be to extend the Pareto Front in the direction of low compression and low $rmse$, because frequently the solutions did not reach this region.
- The second issue is improving the performance for the biggest datasets. Comparison with other instance selection methods showed that for small and medium size datasets our methods has the greatest advantage over those methods. However, for the largest datasets in half of the cases our method achieved lower 1-R^2 and stronger compression with 8-NN than the DROP3-RT method, but in the other half of cases only stronger compression and lower 1-R^2 was produced by DROP3-RT. Our method optimized $rmse$ and 1-R^2 was used only for comparison and the relation between $rmse$ and 1-R^2 is not linear, so it possible that using 1-R^2 as the criterion in the same way as the DROP3-RT used would produce better results. The same can be said of DROP5 ensembles for instance selection in classification tasks. However, because for smaller dataset the advantage of our method even with optimization of different error measure was very significant, our first approach to further improvement of the results on big datasets will focus on a balance between the global exploration and local exploitation.
- The third improvement is reducing the complexity and running time of the algorithm. The complexity for big datasets is now determining by the time of distance matrix calculation.

Chapter 10
Additional Enhancements

Abstract In the previous two chapters we used single or multi-objective genetic algorithms for instance selection. In this chapter we discuss some improvements, which make the algorithms more complex, but allow for obtaining better results (lower Pareto front). Regarding the calculation time, some of the improvements shorten it significantly, while others extend it. However, the main idea of the chapter is to discuss the improvements of the process results and the problem of computational complexity is discussed in the next chapter.

10.1 Introduction

Multi-objective genetic algorithms, and especially NSGA-II, are very good in application to instance selection, however, they are not free of some shortcomings. Here we want to address two problems, which were most evident in our research: decrease in solution quality for very long chromosomes and Pareto front contraction.

The main mechanism to obtain new solutions in genetic algorithms is crossover, when a new child is created by concatenating parts of chromosomes from its parents. The split points between the parts are set randomly. Unfortunately, when the chromosome is very long (thousands of positions or more) it becomes more and more difficult to find such split points and such donors of particular parts of the chromosome, that improving one part will not cause at the same time decreasing the quality of the other part. Although the problem is known, in many of the applications of genetic algorithms it is not such a big issue, because the optimization can run longer with the hope that the proper solution will be finally found. However, in instance selection it is a problem, as the optimal solution here is not the one performing best on the training set, but the one with best generalization capabilities and running the process too long is likely to cause over-fitting.

The natural tendency of multi-objective genetic algorithms (not only of NSGA-II) is to find most of the solutions which satisfy all criteria in a significant degree. As a matter of fact, mostly the crossover operator is responsible for that, as when randomly very different individuals generate a new child, the child will obviously tend to be to a certain degree the average of them. Thus we do not get solutions at the edges of the objective space, and there are no solutions with very strong compression and no solutions with very week compression. The lack of solutions with very strong compression is not a problem, because those solutions also have very low prediction quality and as a result we are not interested in them. However, the lack of week compression solutions may be a problem, because in many cases these solutions will have very good prediction quality, as only a few very noisy instances were removed from them.

In this chapter we discuss the approaches to both of the problems and show how they can help to obtain better quality results.

10.2 Data Space Partitioning

For big datasets, there is a problem with determining the optimal stopping point, as in some areas of the dataset the optimization converges faster than in others. Goldberg wrote that evolutionary algorithms work "by building short, low order, and highly fit schemata (blocks), which are recombined (crossed over), and re-sampled to form strings of potentially higher fitness" [1]. In this way, the over-fitting already may start occurring within such a block, while in other parts of the chromosome still more iterations are required to approach the optimal point. Another issue is that genetic algorithms frequently are less efficient if the chromosome is too long (ten of thousands of positions), that is they require more iterations to converge.

To remedy this problem, data partitioning can be used. It allows to run the optimization in particular areas of the dataset for an optimal number of iterations. Data partitioning is a more effective solution in the regression problems. That is because in regression problems the changes tend to be more smooth and more equally distributed in data space. In classification problems it is important to preserve the decision boundaries and while partitioning the data space we must take a special care not to split the data along or close to the decision boundaries, because instance selection will simply not work withing such partitions. So designing the possible splits would be a complex task and we have not undertaken this research topic yet. Thus, as for now we present only dataset partitioning in instance selection for regression tasks.

Moreover, using the data partitioning, we do not need to calculate the entire distance matrix used by k-NN, but only the distances to several nearest neighbors to each instance. How can we know, which neighbors are nearest, before we calculate the dis-

tance to them? Here we can use some clustering, to split the data into several clusters and then calculated the distance only withing the clusters.

We use the k-means clustering to partition the N instances of the training set into C sets (clusters) $S_1, S_2, \ldots, S_c$ in order to minimize the within-cluster sum of squares (or variance):

$$\arg\min_{\mathbf{S}} \sum_{c=1}^{C} \sum_{\mathbf{x} \in S_c} \|\mathbf{x} - \boldsymbol{\mu}_c\|^2 = \arg\min_{\mathbf{S}} \sum_{c=1}^{C} |S_c| \operatorname{Var} S_c \tag{10.1}$$

where c is the number of the cluster. K-means is a well known algorithm and its description can be easily found in the literature [117] so we do not describe it here. However, it is worth mentioning, that we use also the output variable of the instances as one of the input variables to the clustering, as this proved to produce more adequate results for the purpose of instance selection.

Of course not each optimization problem can be partitioned in this way, because frequently there are mutual interactions between many positions in the chromosome. However, in the case of predicting the output in regression tasks it can be done, because there are no interactions between very remote instances.

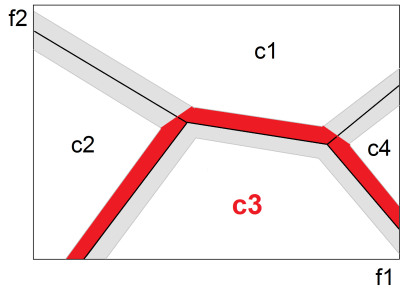


Fig. 10.1. The exact and the extended clusters. For example: the base cluster $c3$ is limited by the black lines. The extended cluster $c3$ contains also the instances in the red zone. The gray zones contain the additional instance of other extended clusters.

Fig. 10.1 shows a dataset with two features $f1$ and $f2$ partitioned into four clusters $c1$, $c2$, $c3$ and $c4$ with the thick black lines. The red zone represents the boundaries of the extended cluster $c3$; this is the cluster itself and all instances from the other clusters that are the nearest neighbors of any instance from the $c3$ cluster.

Algorithm 10 Evolutionary instance selection with dataset partitioning

Partition the $m-th$ training T_m set into C clusters
Optionally determine the extended clusters E for each of the C clusters
for c=0 ... C **do**
  Generate initial currentPopulation of P individuals
  calculate fitness f (Eq. 11.1,11.2,11.3) for currentPopulation individuals
  for i=0 ... $numIterations$ **do**
    apply the crossover operation to generate the newPopulation of P individuals
    calculate fitness f (Eq. 11.1,11.2,11.3) for newPopulation individuals
    sort together currentPopulation and newPopulation individuals by fitness
    select the best P individuals into currentPopulation
    apply the mutation operation
  end for
end for
Merge the selected instances from all C clusters to form the final reduced training set T_{mr}
calculate the $retention_m$ of the training set T_{mr}
calculate the $rmse_m$ on the corresponding test set S_m using the predictor trained on T_{mr}

This approach allows usually for improving both criteria: obtaining lower retention with slightly lower $rmse$, but definitely the retention gets more improved than $rmse$. This is caused by the "boundary effect"; this is we can observe some problems with the instances that are so close to the cluster borders that part of their nearest neighbors belong to another cluster. When we want to predict the output of such an instance using only the neighbors from its own cluster, the results are inadequate. This causes that we cannot further decrease $rmse$ using only that approach. Moreover, the decreasing prediction error inside the clusters is partially canceled out by the increasing error on the borders of the clusters. As a result the optimal size of the cluster exists, which in our experimental evaluation ranged on average from 500 to 1.000 instances.

In order to further improve the results, in [118] we proposed to use a pair of datasets for each cluster: the dataset that is the current cluster T_c and the extended dataset T_e which consists of the current cluster T_c and all k nearest neighbors of all instances from T_c, no matter to which cluster they belong, as shown in Fig. 10.1. We perform instance selection only within T_c, so only the instances from T_c are considered for removal. However, while predicting with k-NN the outputs of instances from T_c (which we need to obtain $rmse$), all the instances from T_e are considered as their potential neighbors. The method is shown in Algorithm 10.

In this work we applied this approach also to the multi-objective evolutionary instance selection for regression tasks and we are happy that the improvement achieved by this approach is even bigger than in the single-objective method.

10.3 Multiply Fronts to Extend Range and Prevent Over-fitting

As the optimization with NSGA-II progresses, the Pareto front gradually extends, as will be explained later. So the first approach in a typical optimization process, where the genetic algorithms optimize directly the final objectives is to increase the number of iterations. But this will be still only a partial solutions to some problems, as the front will not extend fully.

Instance selection belongs to a different class of problems, because, we perform the optimization on the training set and the test set is unknown during the optimization, yet one of the objectives is $rmse$ on the test. Thus increasing the number of iterations will likely cause over-fitting. To prevent over-fitting we have to use early stopping (which can be also implemented by observing she error on a validation set). However, the two conditions are mutually exclusive. To clarify this, we will explain the process of Pareto front formulation during the NSGA-II optimization.

At the beginning of the optimization, all positions in all chromosomes have random values. As a result in the $rmse$ - compression space all individuals (all datasets) are located very close to each other (although special initialization techniques, which were discussed, can partially mitigate this problem), and all have relatively poor balance of $rmse$ - compression. This is shown in Fig. 10.2-a, where the points in blue represent the dominated solutions on the training set, the green points connected with the green line represent the Pareto Front on the training set. Each solution (each training set of selected instances) on the Pareto front is used to train the model. The result of the model prediction on the test set is marked with the corresponding orange point (both points: the orange and the green use the same training set of selected instances, hence they have the same compression value - the compression of the selected training set).

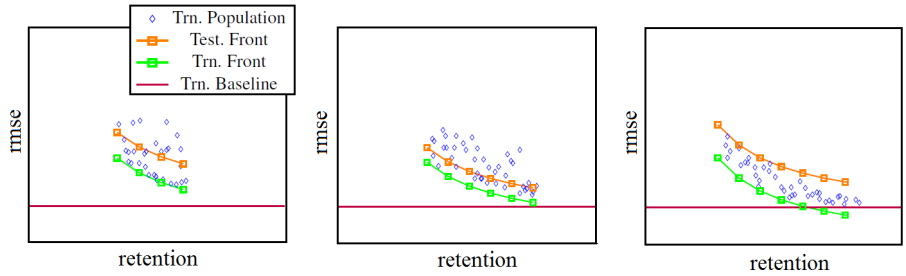
The baseline represented by the purple line in Fig. 10.2 is the $rmse$ on the training set obtained by the regressor trained on the original dataset (without instance selection).

As the optimization progresses, the points move gradually to the positions shown in Fig. 10.2-b and then to Fig. 10.2-c. But before they reach the positions in Fig. 10.2-c, already the over-fitting starts to occurs (the green line in Fig. 10.2-c is lower than in Fig. 10.2-b, and the orange line is situated higher).

However, when we use more fronts, we can reach the solutions with low compression and low $rmse$ with fewer iterations before the over-fitting starts to occur as in Fig. 10.3-c.(the green line is lower in Fig. 10.3-c is lower than in Fig. 10.3-b, and the orange line is also lower).

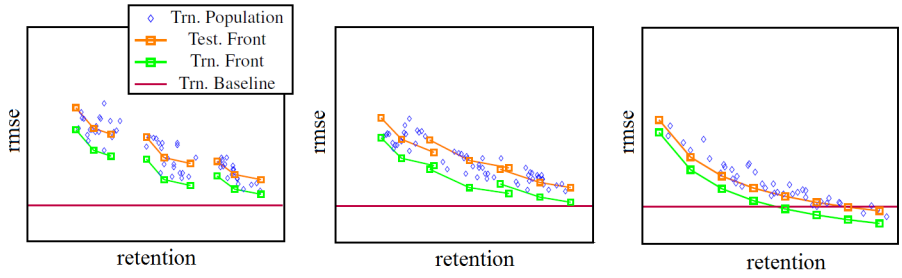
Frequently we do not need the front to be extended in the direction of low compression (high retention rate), because the lowest $rmse$ is already reached below the baseline in Fig. 10.3-b and it will be probably not possible to improve it anymore. In these cases it is enough to generate only the first Pareto Front (and that was the case in about 50% of our experiments).

However, if the lowest $rmse$ is at or above the baseline, we may want to search for a solution with still lower $rmse$. In this case we need to obtain also the other fronts, which we call sub-fronts, to cover a broader space without over-fitting. The sub-fronts are obtained by generating the initial populations with various probabilities of 0 and 1, and then, in the mutation operator, using different probabilities of changing 0 to 1 and 1 to 0 so that the percentage of ones and zeros in the chromosomes remain relatively constant. Finally the sub-fronts will be merged into one Pareto Front (and thus some points from some sub-front may not be included in the final front, if the points from another sub-front satisfy better both objectives).



(a) Initial population (b) The front extends (c) Overfitting

Fig. 10.2. Forming of a single Pareto front during instance selection process.



(a) Initial population (b) The fronts extend (c) No overfitting

Fig. 10.3. Forming of a Pareto front using three sub-fronts during instance selection process.

The whole instance selection process consists of the following steps:

1. Optionally split the data into partitions and perform the remaining steps independently inside each data partition.
2. The distance matrices of the training set are computed and sorted, as discussed in the previous chapters.
3. The population is randomly generated with an average 0.5 probability of 0 and 1 at each position (the probability of each instance being selected and rejected is approximately equal).
4. NGSA-II (or other multi-objective genetic algorithm) is used as the search engine in the solution space with all the tunings and improvements discussed in the previous chapters.
5. The $rmse$ on the training set is evaluated to determine if there is a need for generating additional Pareto fronts. If the $rmse_{trn}$ is lower than 90% of the $rmse_{trn}$ before instance selection or classification accuracy on the training set is 100% then the process terminates. If $rmse_{trn}$ is higher or accuracy is lower, then the following steps are performed (optionally the steps below can be performed also if $rmse_{trn}$ decreased more than 10%):
   a. The population is randomly generated with 0.2 probability of 0 and 0.8 probability of 1 at each position (the probability of each instance being selected is 0.8 and being rejected is 0.2)
   b. NGSA-II (or other multi-objective genetic algorithm) is used as the search engine in the solution space with all the tunings and improvements discussed in the previous chapters. An unsymmetrical mutation operator is used to keep the approximate proportions of selected and rejected instances. The chance of mutation from 0 to 1 is four timer higher than from 1 to 0.
   c. The $rmse$ on the training set is evaluated to determine if there is a need for generating additional Pareto fronts. If the $rmse_{trn}$ is lower than 90% of the $rmse_{trn}$ before instance selection or classification accuracy on the training set is 100% then the process terminates. If $rmse_{trn}$ is higher or accuracy is lower, then the following steps are performed (optionally the steps below can be performed also if $rmse_{trn}$ decreased more than 10%):
      i. The population is randomly generated with 0.08 probability of 0 and 0.92 probability of 1 at each position (the probability of each instance being selected is 0.92 and being rejected is 0.08)
      ii. NGSA-II (or other multi-objective genetic algorithm) is used as the search engine in the solution space with all the tunings and improvements discussed in the previous chapters. An unsymmetrical mutation operator is used to keep the approximate proportions of selected and rejected instances. The chance of mutation from 0 to 1 is 11.5 timer higher than from 1 to 0.
6. All the Pareto fronts are merged together (one, two or three fronts - depending if the two additional populations were generated) to form a single Pareto front.

7. There are two options if data partitioning was used. The first option: the selected instanced from corresponding regions of the Pareto front are merged into single datasets. The second option: multiply final predictive models are used; each trained on one partition of the data and the test vectors are processed by the model trained on the partition closest to the test instance.

10.4 Other Solutions from Literature

Partitioning the search space can be useful to prevent splitting the already created blocks in genetic algorithms and some researchers tried this approach. In [119] a differential evolutionary algorithm with space partitioning was implemented by dividing the search variables into groups of partitions, so each partition contained a certain number of variables and was manipulated as a subspace in the search process. In [120] the search space partitioning was applied to multi-objective genetic algorithms.

Partitioning of the training set was applied to instance selection by de Haro-García and García-Pedrajas [32]. However, the problem with this method is that its performance is degraded for some problems. For the case of instance selection using a genetic algorithm, the authors found that the evolutionary algorithm was too conservative when applied to subsets of the datasets. Thus, many useless instances where retained [32]. This problem was addressed with a recursive application of the stratified approach. After a first application of the stratification and the evolutionary algorithm to the different strata, a new round of stratification was applied with the selected values. This recursive approach was applied until a certain criterion was met. Although this method works for instance selection, its not clear if it could be applied to other problems.

Czarnowski [121, 122] introduced an instance selection method that incorporates several ideas. First clustering was performed on the data and then, within the clusters, the selection was executed by a team of agents. The agents cooperated by sharing a population of solutions and refined the solutions using local search.

One step forward in sampling techniques was developed for instance and feature selection by García-Osorio et al. [65]. This method combines the divide-and-conquer approach of sampling with the combination principle of the ensembles of classifiers. The method is composed of three basic steps: (i) divide the problem data into small disjoint subsets, (ii) apply the learning algorithm of interest to every subset separately (iii) combine the results of the different applications. The first two steps are repeated several times. As the learning method is always applied to small datasets, the the method is quite fast. As the method combines results of the application of the same learning models to different subsets of the available dataset, it was called by the authors democratization of algorithms.

10.5 Experimental Evaluation

W present the results obtained with and without data partitioning and merging up to three Pareto fronts if required. The experimental process is almost the same as shown in Figs. 13.1 and 13.2 with the difference that the test block contains either k-NN or an MLP algorithm and not the four algorithms shown in those figures. For single-objective algorithms we present the results from our work [118] only for the regression datasets larger than 1,500 instances (Tables 10.1 and 10.5), as there was the biggest improvement for these datasets. For multi-objective algorithms we present the results for most of the regression datasets from the Keel repository [8]. The base results come from our work [105] and the results with data partitioning are presented here the fist time. The symbols in the tables: $rr1 = r1/r0$, $rr1p = r1/r0$ with data partitioning, $rr2 = r2/r0$, $rr2p = r2/r0$ with data partitioning, $rr3 = r3/r0$, $rr3p = r3/r0$ with data partitioning. For explanation of the symbols $r0, r1, r2, r3$ see chapter 9, section 9.4.

Table 10.1. Experimental results for single-objective algorithms: inner evaluation algorithm: 1-NN, final prediction model: 1-NN, r0 - $rmse$ without instance selection, r1, r2 - $rmse$ with instance selection without data partitioning, r1p, r2p - $rmse$ with data partitioning for α=0.9 and 0.8 (lower is better), c1, c2, c1p, c2p - corresponding retention rates (lower is better).

dataset	r0	r1	r1p	c1	c1p	r2	r2p	c2	c2p
wankara	0.225	0.222	0.209	0.679	0.623	0.290	0.266	0.147	0.136
plastic	0.617	0.542	0.513	0.718	0.674	0.607	0.589	0.202	0.192
quake	1.344	1.153	1.134	0.672	0.607	1.342	1.296	0.148	0.133
anacalt	0.227	0.232	0.211	0.482	0.470	0.282	0.280	0.194	0.171
abalone	0.915	0.768	0.747	0.697	0.655	0.872	0.844	0.163	0.154
delta-ail	0.716	0.630	0.629	0.618	0.593	0.744	0.740	0.141	0.135
puma32h	1.212	1.025	1.004	0.673	0.644	1.202	1.207	0.165	0.154
compactiv	0.254	0.295	0.295	0.434	0.414	0.307	0.307	0.236	0.216
delta-elv	0.828	0.708	0.710	0.663	0.623	0.838	0.819	0.136	0.128
tic	1.366	1.130	1.118	0.704	0.681	1.330	1.312	0.148	0.142
ailerons	0.657	0.585	0.580	0.596	0.583	0.701	0.688	0.142	0.139
pole	0.244	0.258	0.258	0.662	0.648	0.349	0.335	0.137	0.131
elevators	0.686	0.641	0.635	0.669	0.651	0.763	0.730	0.173	0.151
california	0.654	0.596	0.582	0.676	0.655	0.711	0.703	0.143	0.135
house	0.872	0.775	0.766	0.645	0.625	0.929	0.924	0.141	0.137
mv	0.210	0.199	0.199	0.728	0.721	0.302	0.300	0.164	0.160
average	0.689	0.610	0.599	0.645	0.617	0.723	0.709	0.161	0.151

Table 10.2. Experimental results for binary instance selection: retention and relative $rmse$ with inner evaluation algorithm 1-NN and final prediction model 1-NN. The symbols are explained at Table 10.1.

dataset	rr1	rr1p	c1	c2p	rr2	rr2p	c2	c3p	rr3	rr3p	c3	c3p
mach.CPU	1.159		0.385		1.212		0.216		1.000		0.914	
baseball	0.878		0.662		0.899		0.559		0.878		0.662	
dee	0.811		0.652		0.918		0.401		0.811		0.652	
autoMPG8	0.904		0.685		1.127		0.155		0.904		0.685	
autoMPG6	1.042		0.698		1.242		0.150		0.985		0.920	
ele-1	0.929		0.678		0.966		0.619		0.929		0.678	
forestFires	0.496		0.344		0.514		0.191		0.496		0.344	
stock	1.175		0.629		1.692		0.155		1.000		0.918	
steel	0.911		0.469		1.154		0.369		0.911		0.469	
laser	1.192		0.467		1.324		0.221		1.000		0.911	
concrete	1.117		0.696		1.441		0.155		1.000		1.000	
treasury	1.259		0.662		1.693		0.159		1.000		0.916	
mortgage	1.284		0.545		1.784		0.287		1.000		0.926	
friedman	0.959	0.959	0.692	0.688	1.288	1.247	0.148	0.155	0.959	0.959	0.692	0.692
wizmir	0.919	0.919	0.693	0.644	1.311	1.280	0.160	0.163	0.919	0.919	0.693	0.677
wankara	0.966	0.947	0.692	0.619	1.285	1.253	0.160	0.154	0.966	0.963	0.692	0.657
plastic	0.879	0.834	0.682	0.617	0.983	0.950	0.212	0.180	0.879	0.871	0.682	0.669
quake	0.856	0.843	0.685	0.599	1.003	0.974	0.157	0.145	0.856	0.843	0.685	0.644
anacalt	1.000	0.908	0.449	0.425	1.260	1.236	0.186	0.157	0.977	0.969	0.887	0.865
abalone	0.845	0.821	0.685	0.629	0.962	0.920	0.155	0.127	0.845	0.838	0.685	0.651
delta-ail	0.877	0.871	0.685	0.635	1.031	1.002	0.159	0.165	0.877	0.869	0.685	0.663
puma32h	0.845	0.826	0.689	0.641	1.000	0.979	0.161	0.138	0.845	0.839	0.689	0.676
compactiv	1.177	1.177	0.429	0.401	1.172	1.131	0.237	0.206	0.926	0.916	0.847	0.825
delta-elv	0.851	0.849	0.685	0.621	1.010	0.976	0.157	0.126	0.851	0.849	0.685	0.658
tic	0.827	0.815	0.685	0.641	0.973	0.958	0.156	0.157	0.827	0.822	0.685	0.648
ailerons	0.889	0.874	0.688	0.653	1.079	1.035	0.160	0.154	0.889	0.882	0.688	0.672
pole	1.055	1.048	0.686	0.649	1.443	1.391	0.157	0.135	1.000	0.991	0.922	0.879
elevators	0.930	0.918	0.687	0.648	1.114	1.076	0.162	0.167	0.930	0.925	0.687	0.646
california	0.911	0.883	0.686	0.611	1.099	1.075	0.158	0.126	0.911	0.897	0.686	0.661
house	0.888	0.875	0.684	0.515	1.061	1.042	0.159	0.144	0.888	0.875	0.684	0.658
mv	0.941	0.936	0.686	0.554	1.387	1.349	0.158	0.148	0.941	0.936	0.686	0.673
average	0.923	0.906	0.659	0.600	1.137	1.104	0.167	0.153	0.905	0.898	0.720	0.695
av-total	0.960		0.627		1.175		0.214		0.910		0.740	

Table 10.3. Experimental results for binary instance selection: retention and relative $rmse$ with inner evaluation algorithm: optimal-k k-NN and final prediction model: optimal-k k-NN. The symbols are explained at Table 10.1.

dataset	rr1	rr1p	c1	c2p	rr2	rr2p	c2	c3p	rr3	rr3p	c3	c3p
mach.CPU	1.053		0.436		1.168		0.213		0.964		0.852	
baseball	0.998		0.450		1.112		0.217		0.998		0.450	
dee	1.008		0.403		1.035		0.229		0.986		0.928	
autoMPG8	1.078		0.458		1.134		0.218		0.978		0.927	
autoMPG6	1.043		0.424		1.074		0.228		0.921		0.944	
ele-1	0.954		0.533		1.002		0.367		0.954		0.533	
forestFires	0.839		0.340		0.840		0.162		0.839		0.340	
stock	1.181		0.486		1.386		0.340		0.990		0.954	
steel	1.022		0.444		1.302		0.224		1.000		0.934	
laser	1.031		0.435		1.280		0.208		0.955		0.926	
concrete	1.054		0.488		1.244		0.193		1.000		0.929	
treasury	1.321		0.486		1.768		0.234		1.000		0.927	
mortgage	1.304		0.489		2.387		0.156		1.000		0.923	
friedman	1.073	1.071	0.472	0.453	1.297	1.256	0.162	0.135	1.000	1.000	0.947	0.932
wizmir	1.060	1.054	0.470	0.440	1.297	1.258	0.214	0.208	1.000	0.988	0.779	0.764
wankara	1.098	1.076	0.497	0.451	1.315	1.261	0.160	0.135	1.000	0.989	1.000	1.000
plastic	0.969	0.961	0.446	0.414	0.996	0.968	0.212	0.182	0.969	0.965	0.446	0.407
quake	0.983	0.960	0.685	0.639	1.001	0.973	0.219	0.213	0.983	0.973	0.685	0.632
anacalt	1.013	1.005	0.440	0.414	1.330	1.269	0.174	0.149	0.988	0.982	0.684	0.646
abalone	1.011	0.991	0.682	0.624	1.068	1.015	0.158	0.128	0.981	0.972	0.801	0.741
delta-ail	1.033	1.016	0.686	0.634	1.078	1.029	0.158	0.160	1.000	0.994	0.835	0.783
puma32h	1.014	1.000	0.461	0.438	1.027	1.020	0.222	0.201	1.000	1.000	1.000	0.940
compactiv	1.209	1.175	0.461	0.421	1.238	1.201	0.275	0.241	1.000	1.000	1.000	1.000
delta-elv	1.016	1.001	0.684	0.624	1.029	0.984	0.225	0.206	1.000	0.991	1.000	0.971
tic	0.980	0.974	0.684	0.634	0.996	0.973	0.186	0.156	0.980	0.968	0.684	0.638
ailerons	1.031	1.014	0.473	0.427	1.092	1.062	0.263	0.236	1.000	0.996	0.934	0.886
pole	1.125	1.105	0.491	0.449	1.219	1.183	0.307	0.288	1.000	0.996	0.947	0.896
elevators	1.040	1.018	0.448	0.409	1.112	1.058	0.226	0.220	1.000	0.996	1.000	1.000
california	1.037	1.020	0.488	0.434	1.083	1.055	0.332	0.293	1.000	1.000	0.944	0.881
house	1.029	1.005	0.483	0.419	1.074	1.022	0.276	0.246	1.000	1.000	0.894	0.849
mv	1.136	1.113	0.492	0.404	1.311	1.249	0.273	0.221	1.000	0.994	1.000	0.933
average	1.048	1.031	0.530	0.485	1.142	1.102	0.225	0.201	0.995	0.989	0.866	0.828
av-total	1.056		0.497		1.203		0.227		0.983		0.843	

Table 10.4. Experimental results for binary instance selection: retention and relative $rmse$ with inner evaluation algorithm optimal-k k-NN and final prediction model: MLP neural network. The symbols are explained at Table 10.1.

dataset	rr1	rr1p	c1	c2p	rr2	rr2p	c2	c3p	rr3	rr3p	c3	c3p
mach.CPU	1.089		0.436		1.261		0.213		1.000		1.000	
baseball	1.048		0.450		1.198		0.217		1.000		1.000	
dee	1.012		0.403		1.153		0.229		0.992		0.928	
autoMPG8	0.948		0.458		1.187		0.218		0.942		0.927	
autoMPG6	0.924		0.424		1.141		0.228		0.914		0.944	
ele-1	0.920		0.533		1.040		0.367		0.917		0.533	
forestFires	0.952		0.340		1.014		0.162		0.940		0.340	
stock	0.958		0.486		1.016		0.340		0.945		0.954	
steel	0.882		0.444		0.885		0.224		0.872		0.934	
laser	0.988		0.435		1.011		0.208		0.975		0.926	
concrete	1.017		0.488		1.097		0.193		1.000		0.929	
treasury	0.970		0.486		1.042		0.234		0.968		0.927	
mortgage	0.975		0.489		1.088		0.156		0.958		0.923	
friedman	0.992	0.983	0.472	0.453	1.159	1.148	0.162	0.135	0.985	0.972	0.947	0.932
wizmir	1.013	1.002	0.470	0.440	1.027	1.013	0.214	0.208	1.000	1.000	1.000	0.764
wankara	0.978	0.968	0.497	0.451	1.004	0.998	0.160	0.135	0.968	0.965	1.000	1.000
plastic	0.990	0.985	0.446	0.414	0.995	0.959	0.212	0.182	0.990	0.983	0.446	0.407
quake	0.992	0.983	0.685	0.639	1.017	0.993	0.219	0.213	0.992	0.982	0.685	0.632
anacalt	0.898	0.895	0.440	0.414	1.096	1.089	0.174	0.149	0.894	0.891	0.684	0.646
abalone	0.998	0.985	0.682	0.624	1.006	1.000	0.158	0.128	0.997	0.988	0.801	0.741
delta-ail	0.997	0.987	0.686	0.634	1.005	0.973	0.158	0.160	0.993	0.992	0.835	0.783
puma32h	1.103	1.093	0.461	0.438	1.283	1.259	0.222	0.201	1.000	1.000	1.000	0.940
compactiv	1.018	1.008	0.461	0.421	1.042	1.003	0.275	0.241	1.000	0.996	1.000	1.000
delta-elv	0.998	0.988	0.684	0.624	1.009	1.000	0.225	0.206	0.995	0.989	1.000	0.971
tic	1.000	0.991	0.684	0.634	1.050	1.043	0.186	0.156	0.993	0.985	0.684	0.638
ailerons	1.018	1.011	0.473	0.427	1.041	1.014	0.263	0.236	1.000	1.000	1.000	0.886
pole	1.038	1.000	0.491	0.449	1.188	1.157	0.307	0.288	1.000	0.991	1.000	0.896
elevators	1.011	0.998	0.448	0.409	1.013	1.013	0.226	0.220	1.000	1.000	1.000	1.000
california	0.997	0.983	0.488	0.434	0.999	0.991	0.332	0.293	0.985	0.976	0.944	0.881
house	1.005	1.005	0.483	0.419	1.017	1.010	0.276	0.246	1.000	1.000	1.000	0.849
mv	1.005	1.001	0.492	0.404	1.009	0.985	0.273	0.221	0.991	0.988	1.000	0.933
average	1.003	0.993	0.530	0.485	1.053	1.036	0.225	0.201	0.988	0.983	0.890	0.828
av-total	0.991		0.497		1.068		0.227		0.974		0.880	

Table 10.5. Experimental results for single-objective algorithms: inner evaluation algorithm: optimal-k k-NN, final prediction model: optimal-k k-NN, symbols are explained at Table 10.1

dataset	r0	r1	r1p	c1	c1p	r2	r2p	c2	c2p
wankara	0.167	0.183	0.171	0.447	0.420	0.217	0.217	0.154	0.146
plastic	0.468	0.452	0.452	0.427	0.398	0.466	0.466	0.219	0.198
quake	1.025	1.009	1.010	0.676	0.611	1.030	1.029	0.211	0.200
anacalt	0.212	0.212	0.211	0.480	0.459	0.274	0.273	0.167	0.163
abalone	0.702	0.708	0.705	0.681	0.640	0.756	0.755	0.141	0.132
delta-ail	0.560	0.575	0.572	0.692	0.585	0.605	0.607	0.167	0.148
puma32h	0.896	0.909	0.911	0.477	0.458	0.910	0.905	0.215	0.206
compactiv	0.231	0.277	0.267	0.507	0.463	0.288	0.285	0.258	0.242
delta-elv	0.610	0.622	0.625	0.700	0.613	0.622	0.624	0.230	0.202
tic	1.015	0.996	0.989	0.680	0.656	1.014	1.009	0.177	0.160
ailerons	0.504	0.519	0.516	0.477	0.469	0.555	0.551	0.255	0.231
pole	0.214	0.236	0.233	0.487	0.478	0.254	0.252	0.303	0.302
elevators	0.559	0.577	0.572	0.465	0.451	0.625	0.620	0.207	0.205
california	0.527	0.549	0.551	0.527	0.516	0.579	0.580	0.336	0.324
house	0.687	0.708	0.708	0.563	0.538	0.727	0.721	0.269	0.261
mv	0.140	0.160	0.160	0.488	0.484	0.187	0.186	0.251	0.242
average	0.532	0.543	0.541	0.548	0.515	0.569	0.567	0.222	0.210
Wilcox. p-value		0.04236		0.00044		0.02382		0.00044	

When we compared the results of the evolutionary-based instance selection with the instance selection performed by the regression version of the DROP-3 algorithm [30] (which according to the comparative study in [102] was one of the best classical instance selection algorithms), then for smaller datasets the differences were huge: the evolutionary method was much better, but for the dataset sizes of about 10.000 instances the differences begun to get much smaller. Introducing the data partitioning and performing the instance selection separately inside each partition again allowed for gaining advantage over the DROP-3 results as for the smaller datasets.

10.6 Conclusions

We discussed two improvements of evolutionary instance selection. The first one is data space partitioning, which is mostly useful in regression problems and big datasets. The second one is generating multiply Pareto fronts, which can be useful in regression and classification problems with any dataset size.

The following detailed conclusions can be drawn from this study:

- As instance selection together with many other problems is performed on the training set, over-fitting can occur if the optimization is run too long. Thus, running the algorithm for more iterations is not a solution to improve the results. However, splitting the data into several optimization problems can be successfully applied. Therefore we believe that this approach can be useful also to other problems, where the search space is big and there is no significant interactions between points being far apart.
- The multi-objective evolutionary instance selection method works as well for classification as for regression problems and the properties of the method are similar for both types of problems. We discussed the methods using mostly regression problems and only shortly presented results for classification problems.
- The obtained results were better when optimizing both criteria: compression and accuracy than for single-objective evolutionary-based instance selection.
- It is optimally if the inner evaluation algorithm is the same as the final predictor. For that reason we used 1-NN as the internal evaluator, when 1-NN was used as the final regressor. We used k-NN with optimal k for the MLP neural network final predictor, as a reasonable trade-off between the accuracy and computational cost of this solution.
- Using instance weighting did not work as well as we excepted. However, we noticed two areas where it can provide better results than binary instance selection: 1. for very high compression it is usually able to achieve lower $rmse$ (although the $rmse$ is in any case relatively high in this region), 2. for noisy datasets, that require high k value in k-NN [7].
- It is easier to reduce $rmse$ with instance selection for more noisy data, with higher $rmse$ without instance selection.
- We did not observe any dependency between the compression properties and the data distribution, nor any significant change of the distribution after instance selection.
- We obtained better results than other 12 instance selection methods for regression, for which we were able to get the detailed results for particular datasets to perform the comparison.
- The experiments proved that multiply Pareto fronts allow for extending the final front in both direction and especially to obtain lower $rmse$ comparing to the base algorithm.
- Data partitioning allowed for the additional reduction of data size and for some additional reduction of $rmse$. For the datasets larger than 1000 instances, where the improvements are especially beneficial, we were able to reduce the dataset size by on average 5.5% and at the same time to reduce the $rmse$ by about 1.5% for single objective algorithm. For multi-objective algorithms the improvement was even bigger.

Chapter 11
Optimization of Evolutionary Instance Selection

Abstract There are several factors, which influence the speed of instance selection based on genetic algorithms. In this chapter we discuss crossover operations, population size, reduction of chromosome length, fitness function, population initialization, generational vs steady state algorithms and accelerating distance matrix calculation.

11.1 Introduction

The factors, which influence the speed of convergence of genetic algorithm based instance selection can be divided into those that are more specific to genetic algorithms and those that are more specific to instance selection. We evaluate both of the groups, starting from the first one.

Genetic algorithms are in many cases able to convert to the optimal solution even without optimal parameters, especially in simple problems, optimized directly on the target. Optimizing the parameters allows for significant reduction of computational cost and in more complex problems (e.g. instance selection) also for finding the best solution. In chapter 7 we discussed the elitism and steady state genetic algorithms. However, we should be aware that on parallel architectures the optimal parameters can be different than those determined for a single CPU core optimization.

Using only one CPU core the convergence time would be approximately proportional to the number of fitness function evaluations. However, with multiply CPU cores available in all modern computers, as well as with GPU implementations, it changes, as many operations can be run in parallel. Moreover, it is not always easy to effectively parallelize evaluation of fitness function and it is much easier to evaluate the fitness of several individuals in parallel, especially in the case of instance selection. That is an important factor in deciding if we should use generational or steady state genetic algorithms.

In generational genetic algorithms, the whole new population is created and then it replaces the old population. Three basic mechanisms of the replacement are possible:

1. The new population entirely replaces the old population.
2. We use elitism, and together with the child individuals, the best predefined number of parent individuals are promoted to the next generation.
3. We sort the children and parents together by their fitness function and the most fitted individuals are promoted to the next generation, no matter if they are children or parents.

It is usually the easiest to organize the program in this way that the fitness function of a single individual is evaluated by one CPU core and for that reason steady state algorithms, although require fewer fitness function evaluations are frequently not the fastest solution, as the implementation may not scale well in parallel.

In steady state algorithms, immediately after a new individual is generated it replaces one of the old individuals, usually either the least fitted or the most similar to the new one. In this way the population changes faster and indeed fewer fitness function evaluations can be required for the whole process [83, 123]. However, this approach is not always effective to parallelize, especially if the fitness function evaluation is fast, because caution must be taken that two threads do not try to modify the same individual at the same time. We can solve it with locks, but then another problem appears, if one individual has already been replaced by the new one, the other thread that wanted to replace the same individual will have to find another one to modify. In this case a simpler and more effective solution can be to use a combination of generational and steady state algorithms. For example, if there are P=96 individuals in the population and our computer has C=24 or 48 CPU cores, we can first generate in parallel C new individuals and then update the population. The number of available CPU cores C can also influence the decision of the population size. In this case the optimal population size will be a multiply of C. In some cases with CPU implementations and in probably all cases with GPU implementation it may happen that the number of available cores is larger than the optimal population size (by optimal we understand the population size that allows for the lowest number of fitness function evaluations) and in such a case it is recommended to increase the population size over the one optimal for a single CPU core processing.

Further we discuss optimization of the process parameters, but as it was already mentioned, depending on the available hardware sometimes the process can run faster not using exactly the optimal parameters that we present, but adjusting them to the hardware architecture.

11.2 Optimization of Genetic Algorithms Parameters

Our purpose was to examine how convergence time depends on several parameters. In the experiments we used datasets from 200 to 40.000 instances.

All the experiments presented in this chapter were performed with our software written in C# using as well single-thread processing as Task Parallel Library on a computer with two Xeon E5-2696v2 processors (24 physical CPU cores with hyper-threading turned off and 64 GB RAM).

11.2.1 Population Size and Multi-parent Crossover

First we evaluated the influence of population size on the required number of fitness function evaluations. The discussion presented in this section holds true as well for single-objective as for multi-objective evolutionary instance selection. Although some differences exist, the tendencies are the same. The required number of fitness function evaluation was determined as the point beyond which the solutions no longer improve. In particular cases the number of evaluations depend on many parameters, however for other parameters fixed the dependencies look like these shown in Fig. 11.1 and 11.2, where they were determined for a single objective algorithm.

In the next series of experiments we determined the optimal number of parents of one descendant. An approximate optimum that we obtained for the number of parents is from one for each ten positions in the chromosome for smaller datasets up to one for each hundred positions for larger datasets. The convergence of generational or mixed genetic algorithms with that many parents was over three times faster than with two parents only. Moreover, it provided a higher diversity in the recombination process, so smaller population was needed. The results are shown in Fig. 11.1.

Then we evaluated the influence of population size on the required number of fitness function evaluations. The optimal population size very slightly increases with the chromosome length in the examined range from 200 to 40.000 instances we can assume it is between 70 and 100 and we used the population of 96 individuals with our 24 and 48 CPU core systems to obtain optimal performance. For a bit smaller populations (e.g. 50 individuals) the average convergence time can be slightly shorter, but the process is less stable: the standard deviation is higher and occasionally the algorithm does not converge at all. The experimental results are shown in Fig. 11.2 with other parameters constant and close to optimal. The area to the left of the black stability line in Fig. 11.2 represents unstable solutions, that is the further to the left of this line the higher the probability that the evolutionary process will not converge to any good solution.

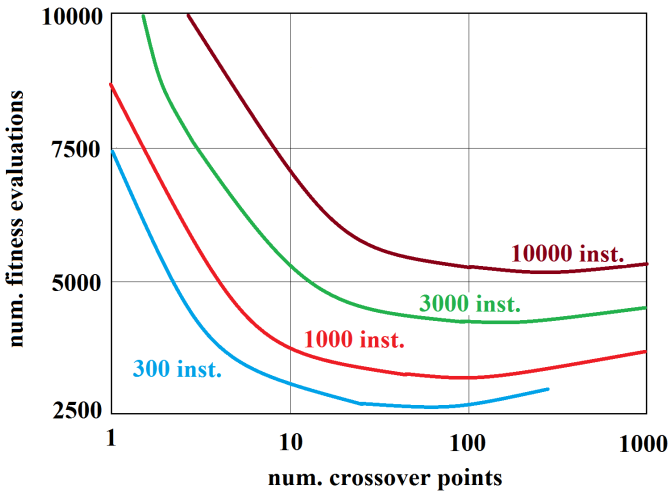


Fig. 11.1. Dependency of the number of fitness function evaluations on the number of crossover points (which in our case equals number of parents - 1) with the population size of 96 individuals for a single-objective genetic algorithms based instance selection.

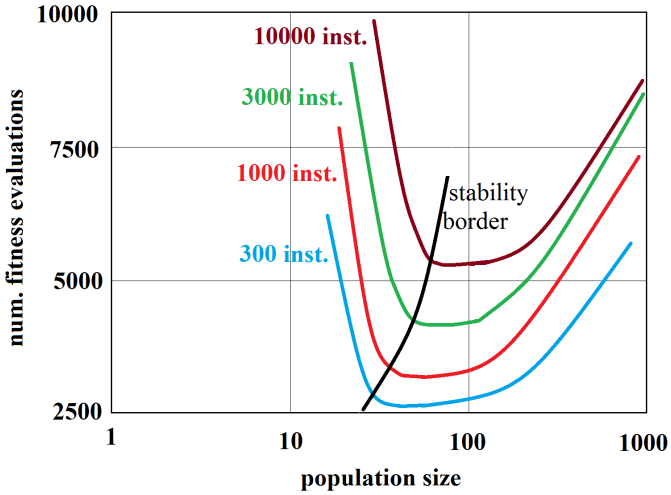


Fig. 11.2. Dependency of the number of fitness function evaluations on population size (number of individuals), with 95 crossover points and 96 parents for a single-objective genetic algorithms based instance selection. The crossover points are randomly selected thus some of them can duplicate.

11.2.2 Fitness Function

As the genetic optimization progresses, the variability among individuals is getting smaller and thus in order to promote the best individuals the fitness function should get steeper. On the other hand too steep fitness function at the beginning of optimization can cause that only the best individuals will take part in generating the offspring and thus the genetic diversity will shortly be very limited with possibly lacking the good chromosome regions that were contained in the generally poor individuals. We designed several different fitness functions, to find out how they perform for instance selection tasks and to propose the optimal solutions depending on the optimization progress [124].

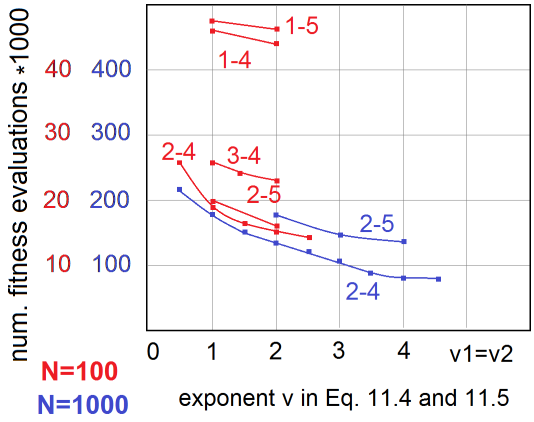


Fig. 11.3. Dependency of the number of fitness function evaluations for 100 (red) and 1000 (blue) instances in the training set (on vertical axis) on the exponents (Eq. 11.1) of the fitness function for different functions for a single-objective genetic algorithms. The numbers (e.g. 2-4) show the number of equation with the function and with exponent formulas: first number (2 in the example) shows the fitness function: 1=Eq.11.1, 2=Eq.11.2, 3=Eq.11.3, second number (4 in the example) shows the exponent function: 4=Eq.11.4, 5=Eq.11.5. 95 crossover points and 96 parents. (source: our work [124])

In Eq. 11.1 the expression before raising to the power v is simply proportional to the accuracy A obtained on the training set **T** and inversely proportional to retention (percentage of selected instances) R in the selected training set S. The average accuracy for the population $avgA$ and average retention $avgR$ only play the role of a

normalizing factor. So this is the most intuitive approach. For regression problems the accuracy in all the equations below can be replaced with the inverse of $rmse_{trn}$.

$$fitness = \Big(\alpha\frac{A}{avgA} + (1-\alpha)\frac{avgR}{R}\Big)^v \tag{11.1}$$

In Eq. 11.2 the expression in the brackets itself makes the fitness function steeper, as now the individual with the minimal accuracy $minA$ has zero fitness in the accuracy part and the individual with the maximum retention $maxR$ has zero fitness in the vector part.

$$fitness = \Big(\alpha\frac{A-minA}{stdA} + (1-\alpha)\frac{maxR-R}{stdR}\Big)^v \tag{11.2}$$

Eq. 11.3 presents the strongest selection, as only the individuals with the quality above the average will have non-zero fitness. Also other cut-off points, situated wherever between zero and $avgA$ (Eq. 11.3) can be used. However, we limited the experiments to the three cases, because it seemed sufficient for drawing the conclusions.

$$\begin{aligned} fitness = \Big(&\alpha \cdot max(0, \frac{A-avgA}{stdA}) \\ &+(1-\alpha)\cdot max(0, \frac{avgR-R}{stdR})\Big)^v \end{aligned} \tag{11.3}$$

The exponent v (Eq. 11.4 and 11.5) can be determined in various ways and we used two approaches. In both approaches it consisted of the constant part $v1$ and the variable part $v2$. In a particular case $v1$ or $v2$ can be set to zero in Eq. 11.4 and 11.5. The variable part in Eq. 11.4 increases linearly during the optimization, where i denotes the current iteration:

$$v = v1 + v2\frac{i}{numIterations} \tag{11.4}$$

In Eq. 11.5 the variable part depends on the speed at which the standard deviation of the fitness expressed by Eq. 11.1 changes, where $stdF(i)$ is the standard deviation of the fitness from Eq. 11.1 in the current i-th iteration and $stdF(i-1)$ and $stdF(i-2)$ in one and in two iterations ago. This ensures that the final fitness function steepness grows as the standard deviation of the population from Eq. 11.1 decreases.

$$v = v1 + v2\frac{0.66\cdot stdF(i-1) + 0.33\cdot stdF(i-2)}{stdF(i)} \tag{11.5}$$

Yet another option is to make $v2$ just inversely proportional to $stdF(i)$.

In the experiments we evaluated 12 combinations (three formulas for the base, each with two formulas for the exponent), each with various $v1$ and $v2$ values and each with two α values ($\alpha = 0.96$ and $\alpha = 0.90$) for instance selection.

The exponent v can also be variable and gradually increase as the optimization progresses. First, we can start from a low v, as $v = 1$ in order not to limit the population diversity and then as the optimization progresses and the individuals tend to be more similar to each other, we gradually increase v to some value, say $v = 6$. In fact this is in many situations the most optimal solution for the reasons presented in Fig. 7.2 and in chapter 7.

11.2.3 Shortening Chromosome

Reducing the chromosome length based on the majority voting at a certain point of the optimization makes the evolutionary instance selection faster. It can be observed that because of the decreasing diversity in the population during the optimization progresses, there are some instances (represented by corresponding positions in the chromosome), which after some number of iterations are selected (1 in chromosome position) by almost every individual and which are rejected (0 in chromosome position) by almost every individual. While the selection of other instances displays greater diversity among individuals. If the optimization continues, the instances (positions) that take at this moment the same values in the great majority of individuals will finally take the majority value with a very high probability, what was experimentally confirmed. Even if it introduces occasionally some local minimum, it is insignificant for the purpose of instance selection, because the difference is only in a few instances and those instances are replaced by others, usually with comparable predictive power. Thus these positions can be at this moment assigned the majority values and removed from the further optimization. That will leave us fewer parameters to optimize and thus the further optimization will be faster. This is depicted in the example below:

```
ind1 0010101010101010010
ind2 0001101010101010110
ind3 0111011001111011110
ind4 0010101010101010010
ind5 1010101010101010010
ind6 0001111110011110000
ind7 0010101010101010010
ind8 0011011111101110110
vote 00????1?1???1?10??0
```

The number in the last row shows the majority position (in our case the position represented in at least 7 out of the 8 individuals). If neither 0 nor 1 is present in at least 7 individuals at the same position, then this position is denoted by a question mark. Thus only the question mark positions will be further optimized and the chromosome can be shortened at this moment. This reduces the number of further iterations as

well as the computational complexity of a single iteration (fewer parents and fewer crossover points have to be generated to produce offspring).

Algorithm 11 The genetic instance selection

```
generate initial currentPopulation of P individuals
for n = 0 ... numIterations do
  apply multiparent multipoint crossover to generate the newPopulation of N
  individuals (N <= P)
  if diversity < Threshold and reducedSearchSpace = false then
    Reduce the search space
    reducedSearchSpace = true
  end if
  if the best solution is kown to be found then
    STOP
  end if
  sort together currentPopulation and newPopulation individuals by fitness
  select the best P individuals into currentPopulation
  apply mutation
end for
```

Also the population size can be decreased, because the optimal population size depends on the chromosome length. However, it grows much slower than the chromosome length, so the factor alone does not reduce the computational cost a lot, but all the three factors together do. There are two parameters, which we adjusted experimentally: the iteration at which the reduction is performed and the voting threshold to eliminate a given position. The pseudo-code of this method is shown in Algorithm 11.

In theory the chromosome length reduction could be performed several times during the instance selection process. In practice it does not make much sense, as little computational time will be gained from the subsequent shortenings and with each such operation the risk that some possible good solutions are eliminated will accumulate.

11.3 Accelerating Calculations of Distance Matrix

The problem was presented in chapter 8, section 8.4 and in chapter 10. Here we will assess the computational complexity of these solutions.

Calculating the distance matrix has a complexity $O(n^2)$ and then sorting it with Quicksort $O(n\text{lon}n)$. For bigger datasets (above about 25,000 instances in our implementation) it is becoming the dominating cost. There are ways to decrease the cost at

the expense of some possible accuracy loss. Nevertheless, the accuracy is more limited by the fact that $rmse$ is evaluated on the training and not on the final test set.

However, as discussed in the previous sections, there is a way to accelerate the first step, by using some of the approximate methods of finding the nearest neighbors. We have chosen to use the K-means clustering with the standard Lloyd's algorithm [125], which has in practice linear complexity in the function of instance numbers. Another solution could be to use the KD-tree space partitioning [126] or other similar methods. As all the methods are approximate, there is some probability that not all the nearest neighbors of each instance will be determined correctly. And to improve this we proposed the extended clusters.

In this way we do not need the entire distance matrix, but only the distances to several nearest neighbors to each instance. We use K-means clustering to cluster the data into several clusters and then calculate the distances only withing the clusters, thus reducing the computational cost almost as many times as the number of clusters, as was discussed in section 10.2 and as is shown in Fig. 11.4. K-means clustering (see Eq. 10.1) has in practice the complexity of about $(C \cdot N)$, where C is the number of clusters. Then the complexity of calculating all the distance matrices only within particular clusters is $O(C \cdot (N/C)^2)$ which is $O(N^2/C)$ (N is the number of instances, C is the number of clusters). In case of using also the extended clusters (see section 10.2) the cost become about 3 times higher. However, because of unequal number of instances in the clusters the complexity is in practice a little higher and using the extended clusters it will grow approximately three times if there are on average two extended clusters used.

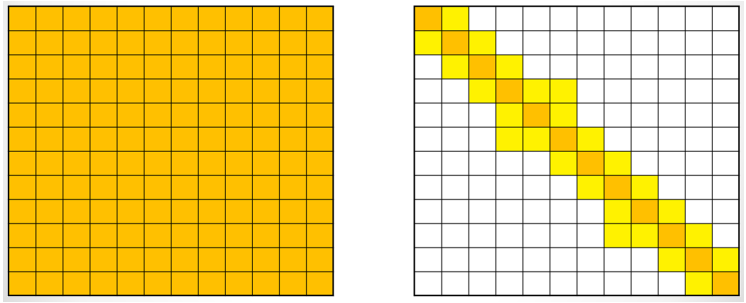


Fig. 11.4. Reduction of operations to calculate distance matrix with data partitioning. On left: calculating the whole distance matirx. On right: in orange - calculating distance only within the base clusters, in yellow - calculating the distances in the extended clusters.

Other options are to use another method of accelerating k-NN calculations, as k-d tree [126] or Locality-sensitive hashing (LSH) [127].

11.4 Analysis of Computational Time and Complexity

In this case, contrary to the popular believe the evolutionary algorithm based solution does not have to be more computationally expensive than the non-evolutionary ones.

The MEISR instance selection process can be decomposed into two steps:

1. The first step - calculating the distance matrices has the complexity $O(n^2)$.
2. The second step - running the evolutionary optimization has the complexity approximately $O(n\log n)$ - because of number of epochs slowly grows with dataset size.

Most of the non-evolutionary instance selection algorithms also must calculate the distance matrix or another equivalent matrix. Their complexity is between $O(n^2)$ (ENN, RHMC, ELH) and $O(n^3)$ (DROP1-5, GE, RNGE) [19]. We really observed that for big datasets the instance selection time with DROP3 grows much faster than with MEISR1 or MEISR2.

In the first step the distances between each pair of instances in the training set are calculated in $O(n^2)$ and then they are sorted using the Quicksort algorithm in $O(n\log n)$, so the complexity of this step is $O(n^2)$. The time spent on calculating the distance matrix also grows with the number of attributes. Therefore not all the points in Fig. 11.5 are situated exactly on the trend lines and the points representing datasets with more attributes are above the trend lines. However, using clustering, we reduce the computational cost approximately as many times as the number of clusters and thus the cost of this step becomes linear if the number of clusters is proportional to the number of instances, for example if we add one cluster for each 500 instances.

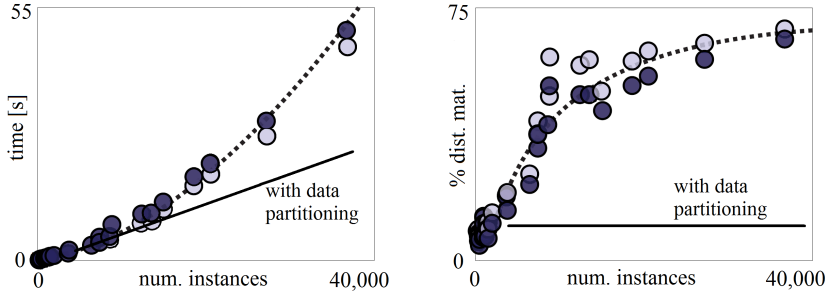


Fig. 11.5. Left: MEISR (version 1 in grey, version 2 with data partitioning in green) running time as a function of number of instances in the original training dataset. Right: Percentage of MEISR running time used to calculate the distance matrix used by k-NN. Light circles denote 1-NN as the inner evaluator and dark circles k-NN with optimal k.

The second step consists of several operations. Calculating the fitness function has the complexity $O(n)$, because the output value of n instances must be obtained, where $n = N$ is the number of instances in the original training set. Obtaining the output value requires reading k non-zero positions from sorted output value arrays, where k is the number of nearest neighbors in the k-NN algorithm, what assuming reduction rate of 50% requires reading $2k$ entries and calculating the average. The time spent in this step grows with the number of k, but much slower than linearly, because also other operations are performed in this step, which do not depend on k, or depend but weaker than linearly, as crossover, mutation, selection. The proportions of time spent at each of them depend on particular software implementation. The experimental measurements confirmed that the complexity of the step can be considered approximately $O(n\log n)$. One operation in the second step takes longer than in the first step, but the number of operations in the second step grows slower with dataset size.

Fig. 11.5 shows the dependency of the MEISR1 running time on the number of instances (left) for 1-NN and k-NN with optimal k and the percentage of the running time used to calculate the distance matrix. The lower lines show the approximate value of MEIRS2 with dataset partitioning. Using also the extended clusters for higher prediction accuracy, the time of calculating the distance matrix is about 3 times longer (using on average two additional clusters, for which the distances to each point must be calculated).

In a practical software implementation, there is also a third-factor consuming time: the constant operations independent of the data size, as calling functions, creating objects, etc. This factor is most significant with very small datasets and this is the reason that the calculation time per one instance is higher for the smallest dataset than for some of the middle size ones. All in all, the analysis of the computational time is very complex and can be done only approximately.

11.5 Conclusions

The purpose of the presented optimization of the evolutionary instance selection process was mostly to decrease the computational complexity, while also taking into account the stability (low variance) of the process. When we originally compared the results obtained with evolutionary based instance selection methods with the results of classical instance selection algorithms, wed obtained better outcome, but on the other hand without the optimizations the calculation time was much longer.

The optimizations discussed in this and previous chapters allowed to shorten the process time and make it comparable with the time of the classical methods. We optimized the number of crossover points, the size of the population and the fitness function. We added the caching of the information required by k-NN and some other improvements.

The first step - calculating the distance matrices has the complexity $O(n^2)$ and sorting them with Quicksort $O(n \log(n))$. With the dataset T size of N=1,000 instances this took about 10% of the whole process and with N=40,000 about 65% with 21 attributes and for larger datasets it becomes the dominating cost. However, the cost of this step can be reduced to linear complexity by using data partitioning. The complexity of the second step - the evolutionary optimization can be assessed only approximately as $O(n \log(n))$, but in any case it is definitely less much than $O(n^2)$.

In the case of instance selection, contrary to the popular believe the evolutionary based solution does not always have to be more computationally expensive then the non-evolutionary ones. Most of the non-evolutionary instance selection algorithms also calculate the distance or equivalent matrix and then perform some other operations with total complexity usually between $O(n^2)$ and $O(n^3)$.

Chapter 12
Joint Evolutionary Feature and Instance Selection

Abstract In this chapter we discussed joint evolutionary instance and feature selection. We present three basic approaches: using sequential feature and instance selection, dividing the chromosome into instance part and feature part and using coevolutionary methods, where feature and instance population are evaluated separately, but use the other population for calculating fitness value.

12.1 Introduction

Instance selection and feature selection have basically the same purpose: to reduce the dataset size and possible to improve prediction quality of the models trained on the reduced dataset. Traditionally different algorithms are used for instance and feature selection due the different concept of instances and features.

Nevertheless, using evolutionary computations more options are at hand. One of the options is to use a sequential approach, either as a traditional feature selection (based on feature filters) followed by evolutionary instance selection or as evolutionary feature selection followed by evolutionary instance selection. Another possible solution is to use part of the chromosome to encode instances and other part to encode features and to perform single evolutionary optimization. Still another option is to apply coevolutionary approach with a separate population of features and instances and periodically merge the best results. Each of these options has some advantages and drawbacks, so in this chapter we will analyze all of the three approaches. Each of the approaches can be implemented in many different ways and we will present some of the implementations in order to give the reader sufficient knowledge to understand the whole problem.

12.2 Sequential Evolutionary Feature and Instance Selection

For instance selection pre-calculating and caching the sorted distances between particular instances is a very efficient method for accelerating the k-NN algorithm used as an inner evaluator within the instance selection process (see chapter 8, section 8.4). For feature selection the method is not fully applicable. Nevertheless, some speed up can be achieved by pre-calculating the distances, but this is much less effective than for instance selection.

To understand this problem, let us assume that there are five attributes (a, b, c, d, e) and let us recall how the square of the distance between two examples $dist(x_1, x_2)$ is calculated in this case:

$$dist(x_1, x_2) = (x_{1a} - x_{2a})^2 + (x_{1b} - x_{2b})^2 + (x_{1c} - x_{2c})^2 + (x_{1d} - x_{2d})^2 + (x_{1e} - x_{2e})^2 \tag{12.1}$$

We do not need to calculate the square root of $dist(x_1, x_2)$, as we need only to find out, which distances are smallest (which are the nearest neighbors) and not to obtain their exact value. Now we can see that if one (or more) feature gets removed, the distance must be re-calculated for each pair of instances. Let us write this equation in a shorter form:

$$dist(x_1, x_2) = dist_a + dist_b + dist_c + dist_d + dist_e \tag{12.2}$$

As we can see from Eq. 12.2 we can pre-calculate the squares of particular components, although for a dataset with many features and many instances we can run out of memory. A solution to fit the arrays into memory is the dataset partitioning discussed in chapter 11. Let as assume, there are N=100,000 instances and F=1000 features. To keep all the partial distances in memory we need to store about $0.5FN^2$ numbers, that is 10e+12, what assuming the double type will give 3.64TB memory, what as of 2019 exceeds the capacity of most single computers and would require some advanced server configurations. But when we use 100 clusters with an average size of 1000 instances, that will sum up to 37GB RAM, what makes this possible to use a typical computer to process this data. Moreover, the calculation time will be also shortened 100 times.

Another proposition of tackling this problem was presented in [29] and we present this solution in section 12.3.

However, there are still some heuristics based on the experiments that can be used to further limit the memory requirements. First, we sort the distance matrix with all features and then we assume that if some features get removed - the order of the sorted distance matrix will remain intact. So for example for 3-NN we update the distance only for 20 nearest neighbors. So in fact we have to calculate the $20N$ distances and not $0.5N^2$ distances and we already have the squares precalculated. That is a significant speed increase comparing to calculating everything from scratch each time, but

definitely this is not so effective as in case of instance selection. Moreover, we never have a guarantee that the closest neighbors will not move beyond the 20th instance. But we can try to predict this and dynamically update the search distance basing on the relations of the first 20 distances. If the minimum distances are at the beginning, there is probably no need to exceed 20 positions in the search. However, if the minimum distances are evenly split over the 20 positions, it is likely that the search space has to be extended. This can be repeated iteratively as long as we clearly depart from the minimum for the further elements of the original distance matrix.

That approach is getting a bit complex, but it saves as well memory as calculation time. The downside of it is that it does not guarantee with 100% certainty that the obtained order of instances will be correct in each case, but also genetic algorithms do not guarantee that the optimal solution will be found. However, in both cases usually the solution enough close to the optimal is found and it is enough to make us happy, especially that the results are usually better than those obtained with non-evolutionary approaches.

Fortunately, there are usually fewer features than instances, so the same fragments are more likely to repeat in various chromosomes, especially at the final stage of the optimization. Also other individuals will not differ too much, so the difference can be added/subtracted to the sum to further accelerate the process.

As discussed in chapter 6, section 6.4 the first stage that should be performed in data selection is noise removal and in typical data more there is a higher percentage of noisy features than noisy instances. Thus in most cases feature selection should be performed first. It can be done either using feature filters, wrappers os evolutionary methods as mentioned in the previous section. The next state is to perform evolutionary instance selection, which in this case is really independent from the feature selection.

12.3 Features and Instances Encoded in the Same Chromosome

In this approach a part of the chromosome encodes instance and other part encodes features. From the evolutionary algorithm view point the features and instances are indistinguishable. That is the simplest case but not necessarily the best one.

The first problem with this approach is definition of the fitness function. Pérez-Rodríguez and others [128] proposed the following fitness function

$$fitness = \alpha_1 \cdot acc + \alpha_2 \cdot (1 - f) + \alpha_3 \cdot (1 - n) \tag{12.3}$$

where acc is the accuracy of the individual measured using a nearest neighbor rule, f is the fraction of selected features, n is the fraction of selected instances and $\alpha_1 + \alpha_2 + \alpha_3 = 1$.

However, in our opinion a better fitness function is:

$$fitness = \alpha \cdot acc + (1 - \alpha)\frac{1}{f^y \cdot n^v} \tag{12.4}$$

This function minimizes directly the dataset size, which is expressed by the product $f \cdot n$, which usually corresponds to the computational cost of predictive model learning. Typically the exponents y and v will be set to 1. However, if there is such a need, f and n can be raised to different powers.

In [129] simultaneous instance and feature selection and weighting using evolutionary computation was discussed. For that purpose real numbers were evolved together with binary values. The authors combined for that purpose differential evolution algorithm and a CHC genetic algorithm. The weights were evolved using a differential evolution algorithm and selection using the binary CHC algorithm. For N instances and F features in the training set, the chromosome had the length of $2N+2F$. For each instance and feature, the weight was encoded in the chromosome and a binary value denoting if it is selected. If any of the four tasks of feature selection, feature weighting, instance selection and instance weighting was not performed, the corresponding part of the chromosome was set to 1 in all positions representing this operation and fixed. Recombination was performed with the standard differential evolution recombination method combined with HUX crossover for the binary part of the chromosome.

In [130] the authors used binary value coding to select feature and instances. The objective function was the composition of the precision of 1-NN plus a minimization of the number of features and instances. They presented the results only on two datasets: Satimage and an artificially generated dataset. The first part of the chromosome encoded features and the next part instances.

Chen [131] presented a multi-objective design to joint feature and instance selection called intelligent multi-objective evolutionary algorithm (IMOEA). The goal was to maximize the accuracy of the 1-NN classifier and minimize both the number of features and instances. Here also the first part of the chromosome encoded features and the next part instances. However, they implemented "intelligent crossover" operator, which uses only two parents, but tests several possible crossover points and chooses the best one.

Ros et. al. [132] applied a multi-objective approach with a two-phase genetic algorithm to select optimal features and instances. The search was optimized by dividing the algorithm into self-controlled phases managed by a combination of pure genetic process and dedicated local approaches. Different heuristics such as an adapted chromosome structure and evolutionary memory were introduced to promote diversity and elitism in the genetic population.

In [133] the authors used a genetic algorithm, which gave precedence to feature selection over instance selection, by using a higher probability to the changes that remove features from the solutions. The first part of the chromosome encodes the F features and the next part the N instances. In the genetic algorithm the parents

were selected randomly without considering their fitness. Next from the population consisting of p parents and p children, p best individuals were selected. Moreover, they use the "biased" mutation, that is a higher probability is assigned to the change from 1 to 0 than from 0 to 1 in the instance part of the chromosome to keep the number of instances low. We used a similar approach in a multi-objective optimization to extend the Pareto Front, as presented in chapter 10.

In [134] an approach was presented that splits chromosome in two parts, one for features weights, which are encoded as real values in the range [0..1] and the other one for instances, which are encoded as boolean values for binary instance selection.

12.4 Coevolutionary Feature and Instance Selection

In [135] and [136] cooperative coevolution was implemented as the co-existence of some interacting populations, evolving simultaneously was presented. Three populations were cooperating to get the best possible solution and each of them was focused on one data reduction task. The first population performs instance selection, the second - feature selection, the third population performed both instance and feature selection. A candidate solution was created by joining individuals chosen from each population. To evaluate each individual a selected individuals from the other populations (so called collaborators) were used and then the collaborators were merged - as in a standard coevolutionary approach.

In [137] the authors applied the Estimation of Distribution Algorithm (EDA), which is a modification of a genetic algorithm, to select instances and features in the problem of estimating the likelihood of one medical dataset. In EDAs, there are neither crossover nor mutation operators, the new population is sampled from a probability distribution which is estimated from the selected individuals. A randomized, evolutionary, population-based search is performed using probabilistic information to guide the search. In this way, both approaches (genetic algorithms and EDAS) perform the same operations, except that EDAs replace genetic crossover and mutation operators by a probabilistic model of selected promising solutions and generating new solutions according to that model.

In [29] a scalable evolutionary simultaneous instance and feature selection algorithm was presented. Is was based on the divide-and-conquer principle and on bookkeeping, which allowed the execution of the algorithm be almost linear in time. The divide-and-conquer approach of applying the selection process to subsets of the whole dataset was used for instance selection in some literature positions and in this work was extended to simultaneous instance and feature selection. Thus random partitions of the instances and features were created and the selection was performed within each of the subsets. To avoid a substantial randomness and its negative effects, this partitioning was repeated for several times and the results were combined by a voting

process. Ensembles obviously stabilize the results, as was already discussed in the first part of the book.

In [138] the authors used two simultaneous Simulated Annealing runs to solve each problem separately but use the actual solution of each process to calculate the quality of both of them. That is to a certain degree similar to the idea of coevolution.

In [76] a hybrid evolutionary algorithm for data reduction, using both instance and feature selection, was presented. A global process of instance selection, carried out by a steady-state genetic algorithm, was combined with a fuzzy rough set based feature selection process, which searches for the most interesting features to enhance both the evolutionary search process and the final preprocessed data set.

Also some other complex hybrid methods were presented in literature ([139, 140, 141]).

12.5 Conclusions

An approach to simultaneous evolutionary feature and instance selection proposed by most authors was based either on splitting the chromosome into feature part and instance part or on different versions of coevolutionary methods. Coevolutionary methods have this advantage over dividing the chromosome into feature and instance part that the chromosomes are shorter and thus the optimization is faster. Nevertheless, the computational cost of such solutions still remains high if there are many features, as accelerating the calculation in case of feature selection cannot be done so effectively as can be done for instance selection by calculating an sorting the distance matrices only once before the optimization. For that reason for very large dataset hybrid solutions based on feature filters with evolutionary instance selection can also be considered, if time limit is the concern.

Chapter 13
Instance Selection for Multi-Output Data

Abstract In this chapter we present instance selection for multi-output data using multi-objective evolutionary algorithms. A multi-target regressor based on k-NN is used as an inner evaluator to assess the error inside the instance selection process, while the final prediction is performed using different multi-target predictive models. In this chapter we incorporate all the solutions described so far, as multiple Pareto fronts, dataset partitioning, proper population initialization and others. The results are presented on the benchmark datasets and they are very promising, showing that the proposed method greatly reduced dataset size and, at the same time, improved the predictive capabilities of the multi-output regressors.

13.1 Introduction

Thus, in this chapter we will focus on instance selection in multi-target regression problems. There are a lot of papers on instance selection for single-label classification and also some, but definitely much fewer for single-label regression. However, there were only a few papers on instance selection for multi-label classification. We have written a paper on this topic [142], which to the best of our knowledge this is the first paper on instance selection for multi-output regression. The experimental results presented in this chapter are based on this paper.

Multi-output classification, also known as multivariate or multi-target classification, is a task of predicting of multiple classes by using a set of input variables. In other words: the inputs are similar as in a single output classification problem, but the output is different, as it consists of many single outputs.

By analogy, multi-output regression, also known as multivariate or multi-target regression, is a task of predicting of multiple continuous values by using a set of input features [143, 144].

The simplest way to predict each of the outputs of the multi-output problem is to independently train a different model for each of the outputs. This approach is called a single-target method. However, the drawback of this method is that this does not use the relationships between the different outputs, which can improve the prediction quality and thus methods, which use these relationships have proven to give better results [145, 146]. There are two groups of approaches to multi-output data: data transformation and algorithm adaptation [145, 147]. The data transformation methods rely on transforming the multi-output label dataset into a set of single-target datasets, which are then used for training a model for each target. The prediction is made by concatenating the different predictions of each regressor. In this chapter, we use four data-transformation methods implemented in the Mulan package [148] - the most popular package for the multi-target problems (`http://mulan.sourceforge.net/datasets-mtr.html`):

- Single-target regressor - the equivalent of the binary relevance method [149] for classification. It creates as many single-output datasets as the number of outputs in the original multi-output dataset - one dataset for each output. The regressor is then trained on each of these sets to predict one output.
- Multi-target stacking - adapts the stacked generalization [150] to multi-target data. It consists of two stages. In the first stage, an independent model is trained for each output. In the second stage the same number of models are used as in the first stage, but the predictions of the first stage models are added to the original attributes.
- Regressor chain [151] - several regressors are chained in sequence, the first one is trained on the inputs only and predicts the first output. The second one is trained on the inputs and the output from the first one. The third one is trained on the inputs and the output from the first and the second one and so on. The drawback of this method is that it depends on the order in which particular regressors are added to the chain. In general they should be added from the best performing to the worst performing one.
- Ensemble of regressor chains - the ensemble combines several regressor chains with different chaining orders into an ensemble. This method has similar advantages as single-output ensembles: minimizing the influence of the regressor order in the chain and improving prediction quality.

Instance selection for multi-output regression is not an easy task, as it comprises two problems: instance selection for regression [34] and multi-target data. Both of the issues are not simple and their combination is especially challenging. Even using instance selection for multi-label classification datasets [152] was not an easy task. This is one of the reasons for which we decided to use the evolutionary approach, as we do not have to explicitly define the rule for particular instance removal.

13.2 Multi-Objective Evolutionary Instance Selection for Multi-output Regression

We adapted the MEISR2 method for multi-target data. In the adaptation process we needed to change two elements: the inner evaluation model inside the instance selection and the final predictor. As the final predictors we use the four models explained in the previous section implemented in Mulan, this is:

1. single-target regressors based on k-NN
2. multi-target stacking
3. regressor chain of k-NN
4. ensemble of k-NN regressor chains

We have implemented in our software used for instance selection two of these models as the inner evaluators: single target regressor and regressor chain. As it can be expected from the experience with instance selection for single target data, the best results can be obtained if the evaluation model inside the instance selection process is the same model as the final regressor.

However, for the sake of speed of the solution we implemented only single target regressor and regressor chain. But the selected sets obtained with the two models were in most cases almost the same. That was caused by the "discrete" properties of the k-NN algorithm. In a regressor chain the output of the previous predictor is added as an additional input of the next predictor. But in order to change the results comparing to single target regressor, at least one of the nearest neighbors in k-NN has to change. However, the change occurred at a very low frequency, so even though it sometimes changed the predicted output for a given target, this very seldom resulted in a different decision about the instance selection or rejection.

The conclusion from this is that in order to achieve the improvement from regressor chain, three conditions should be satisfied:

1. The targets should be sorted in increasing order of the $rmse$ of a single target predictor.
2. The prediction model should not be k-NN, but a "continuous" model, which will always change its output if one additional input is added (as linear regression for instance).
3. The task should be rather regression or instance weighting than classification or binary instance selection, as in regression the change is directly applied and in classification or binary instance selection the decision about a given instance changes only if the change from the previous point is big enough to exceed the threshold.

Using any other base models than k-NN, would be very time consuming, as it was already discussed. We finally decided to use only single-target regressors and regressor chains based on k-NN as the inner evaluator.

The process was performed using the NSGA-II as the search engine in a way as described in chapter 9 with generation of the additional Pareto fronts as needed and the dataset partitioning if the number of instances is greater than 1500, as described in chapter 10.

13.3 Experimental Evaluation

The performance of the presented instance selection for multi-output data was experimentally evaluated in 10-fold cross-validation using multi-output regressors and compared the results with the prediction obtained by the multi-target regressors trained on the original, uncompressed datasets [142].

The experiments were carried out on the 13 multi-output regression datasets (see Table 13.1) that are the benchmark files available from the Mulan project website. There were 14 benchmark datasets available at the Mulan website. The biggest one River flow 2 had the same number of instances as River flow 1, but 8 times more attributes. Although our software was able to perform the instance selection on that dataset, the Mulan package that we used for the final predictions, went out of memory with the 64 GB RAM that we had in our computer. So only the results for the 13 datasets are reported.

Table 13.1. Summary of datasets characteristics: name, domain, number of instances, features, and outputs.

Dataset	Domain	Instances	Attributes	Outputs
Andromeda	Water	49	30	6
Slump	Concrete	103	7	3
EDM	Machining	154	16	2
ATP7D	Forecast	296	211	6
Solar flare 1	Forecast	323	10*	3
ATP1D	Forecast	337	411	6
Jura	Geology	359	15	3
Online sales	Forecast	639	401	12
ENB	Buildings	768	8	2
Water quality	Biology	1060	14	16
Solar flare 2	Forecast	1066	10	3
SCPF	Forecast	1137	23	3
River flow 1	Forecast	9125	64	8

13.3.1 Experimental Setup

The experimental setup is presented in Fig. 13.1 and 13.2. As there were some missing data and nominal attributes in the original datasets, two additional preprocessing steps were performed: missing data imputation by replacing missing values by the mean value of the feature, and nominal attributes replacement by transformation into binary attributes with the methods implemented in the $Weka$ package.

The NSGA-II genetic algorithm used the following parameters: 25-35 epochs for each dataset (more for larger datasets), 96 individuals in the population, and the crossover, selection and mutation operations performed as described in the previous chapters. Also the distance matrix was prepared in a similar ways, as in the case of multi-objective instance selection for single output regression, however with the difference that the values for all outputs were stored.

Only the first Pareto front was needed in over half of the experiments, as the lowest $rmse$ obtained in the first front on the training set was already below 95% of the baseline and the additional sub-fronts permitted no further lowering of the $rmse$ and therefore were not included in the final front.

In the testing part, the base regressor was k-NN with k = 1, 3 and 5, adapted to multi-output regression by four different techniques [153]: single-target regressor, multi-target stacking, regressor chain, and ensemble of regressor chains. All of the parameters of the regressors were set to the default values in Mulan. The average root mean squared error ($rmse$) in the multi-output version was used as a measure of the prediction quality:

$$rmse = \frac{1}{t}\sum_{j=1}^{t}\sqrt{\frac{\sum_{i=1}^{N_{tst}}\left(y_{ij}-\hat{y}_{ij}\right)^2}{N_{tst}}} \tag{13.1}$$

where, N_{tst} is the number of instances in the test set, t the number of targets (outputs), y_i and $\hat{y}_i$ are respectively the actual and predicted outputs for the instance $\mathbf{x}_i$.

We report the results for three characteristic points:

- $(c(r_{min}), r_{min})$: The first point of the Pareto front, i.e., the solution with the lowest compression and with the lowest $rmse$ on the training set. Although it cannot be guaranteed that the model trained on this subset will produce the lowest $rmse$ on the training set in every case, it will usually do so.
- $c(r_{base})$: The point of the Pareto front when the front intersects the baseline on the test set (we understand by the baseline the $rmse$ obtained on the test set with the regressor trained on the whole training set).
- $(c(r_{105}), r_{105})$: The point of the Pareto front that achieves an $rmse$ on the test set 5% higher than the first solution $(c(r_{min}), r_{min})$.

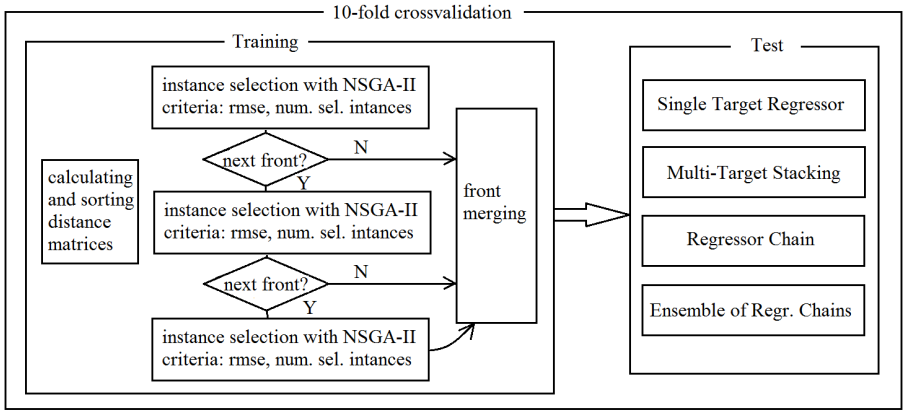


Fig. 13.1. The experimental process without data partitioning.

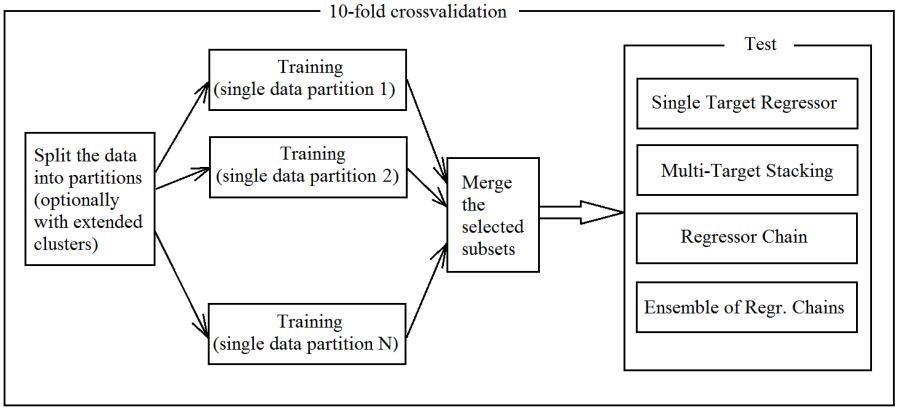


Fig. 13.2. The experimental process with data partitioning. The "Training" (single partition) block is shown in Fig. 13.1.

13.3.2 Experimental Results

All the reported results are the averages in 10-fold cross-validation. Tables 13.2, 13.3, and 13.4 show the $rmse$ obtained with the k-NN regressor-based single target, multi-target stacking, chain of k-NN, and an ensemble of k-NN chains: r_{min} - the minimal $rmse$ obtained with instance selection, $r0$ - $rmse$ without instance selection (the models were trained on the whole original dataset).

Table 13.5 shows the compression (in percentage) achieved by the instance selection methods with k=1, 3 and 5, at the minimum $rmse$ obtained with instance selection (r_{min}), at the baseline $rmse$ ($r0$) and at r_{105} ($rmse$ 5% over the r_{min}).

The solutions obtained on the test sets also frequently form a Pareto front corresponding to the Pareto front on the training sets. The ensemble of regressor chains displays relatively low dependence on instance selection. That is, it is difficult to improve the $rmse$ with instance selection, but on the other hand quite a strong instance selection is possible without degrading its performance. The biggest improvements in $rmse$ were observed for the models, which used 1-NN. That can be explained in the same way as it was explained for a single output data, that removing the nearest neighbor in that case change the results very strongly.

Table 13.2. Summary of the results for $rmse$ obtained with multi-output k-NN regressors with k=1 (lower is better).

	s.targ.		stacking		chain		ensemble	
Dataset	$r0$	r_{min}	$r0$	r_{min}	$r0$	r_{min}	$r0$	r_{min}
Andromeda	0.447	0.441	0.447	0.441	0.447	0.441	0.411	0.491
SCPF	0.958	0.763	0.958	0.763	0.958	0.763	0.826	0.745
Water quality	0.942	0.894	0.942	0.894	0.942	0.894	0.828	0.798
Solar flare 1	1.225	0.898	1.231	0.898	1.252	0.897	1.108	0.892
Solar flare 2	0.935	0.839	0.950	0.838	0.979	0.838	0.923	0.816
Slump	0.908	0.803	0.908	0.803	0.908	0.803	0.751	0.786
ATP1D	0.553	0.515	0.553	0.515	0.553	0.515	0.477	0.555
ATP7D	0.782	0.734	0.782	0.734	0.782	0.734	0.681	0.670
EDM	0.603	0.525	0.603	0.525	0.603	0.525	0.580	0.522
River flow 1	0.090	0.066	0.090	0.066	0.090	0.066	0.083	0.065
ENB	0.573	0.525	0.573	0.525	0.573	0.525	0.462	0.417
Jura	0.813	0.805	0.813	0.805	0.813	0.805	0.730	0.743
Online sales	0.900	0.858	0.900	0.858	0.900	0.858	0.790	0.778
Average	0.748	0.667	0.750	0.667	0.754	0.666	0.665	0.637

Table 13.3. Summary of the results for $rmse$ obtained with multi-output k-NN regressors with k=3 (lower is better).

	s.targ.		stacking		chain		ensemble	
Dataset	$r0$	r_{min}	$r0$	r_{min}	$r0$	r_{min}	$r0$	r_{min}
Andromeda	0.587	0.506	0.564	0.466	0.570	0.497	0.599	0.528
SCPF	0.843	0.736	0.857	0.735	0.833	0.735	0.776	0.737
Water quality	0.788	0.780	0.787	0.785	0.789	0.787	0.765	0.765
Solar flare 1	1.027	0.909	1.060	0.908	1.018	0.910	1.012	0.892
Solar flare 2	0.907	0.859	0.905	0.868	0.887	0.859	0.914	0.851
Slump	0.700	0.739	0.686	0.770	0.720	0.750	0.716	0.724
ATP1D	0.438	0.457	0.439	0.458	0.437	0.457	0.431	0.444
ATP7D	0.626	0.636	0.626	0.636	0.624	0.634	0.608	0.606
EDM	0.586	0.583	0.571	0.580	0.574	0.583	0.627	0.616
River flow 1	0.092	0.071	0.093	0.072	0.093	0.071	0.075	0.074
ENB	0.296	0.304	0.276	0.296	0.291	0.308	0.343	0.325
Jura	0.722	0.727	0.735	0.738	0.729	0.723	0.724	0.727
Online sales	0.800	0.819	0.798	0.824	0.794	0.815	0.778	0.782
Average	0.647	0.625	0.646	0.626	0.643	0.625	0.644	0.621

Table 13.4. Summary of the results for $rmse$ obtained with multi-output k-NN regressors with k=5 (lower is better).

	s.targ.		stacking		chain		ensemble	
Dataset	$r0$	r_{min}	$r0$	r_{min}	$r0$	r_{min}	$r0$	r_{min}
Andromeda	0.587	0.601	0.564	0.580	0.570	0.610	0.599	0.621
SCPF	0.785	0.731	0.789	0.740	0.759	0.728	0.737	0.730
Water quality	0.770	0.772	0.773	0.777	0.776	0.780	0.762	0.766
Solar flare 1	0.953	0.904	0.989	0.906	0.956	0.906	0.948	0.908
Solar flare 2	0.886	0.842	0.905	0.861	0.889	0.845	0.894	0.845
Slump	0.712	0.709	0.715	0.718	0.727	0.721	0.706	0.705
ATP1D	0.443	0.441	0.442	0.443	0.442	0.441	0.432	0.438
ATP7D	0.610	0.638	0.610	0.637	0.609	0.639	0.599	0.643
EDM	0.581	0.570	0.584	0.570	0.584	0.570	0.634	0.675
River flow 1	0.087	0.075	0.081	0.076	0.087	0.075	0.079	0.079
ENB	0.312	0.303	0.311	0.299	0.309	0.305	0.318	0.312
Jura	0.722	0.725	0.735	0.736	0.729	0.730	0.724	0.734
Online sales	0.811	0.801	0.805	0.801	0.806	0.800	0.781	0.791
Average	0.635	0.624	0.639	0.626	0.634	0.627	0.632	0.634

Table 13.5. Summary of the compression results (in percentage) of the instance selection method.

	k=1			k=3			k=5		
Dataset	r_{min}	r_{base}	r_{105}	r_{min}	r_{base}	r_{105}	r_{min}	r_{base}	r_{105}
Andromeda	45.12	45.12	49.29	22.68	30.55	40.74	17.14	17.14	17.80
SCPF	67.99	82.39	82.39	67.16	83.18	86.35	67.62	75.20	94.14
Water quality	51.32	80.86	80.86	62.41	66.21	80.31	9.84	9.84	81.08
Solar flare 1	69.39	89.82	95.51	82.64	89.65	89.65	67.67	83.02	89.48
Solar flare 2	69.55	82.46	82.97	66.47	78.85	78.85	72.49	82.85	82.85
Slump	56.73	78.85	80.05	54.14	54.14	56.31	19.61	19.61	63.73
ATP1D	49.29	52.30	59.94	49.31	49.31	60.15	10.68	10.68	57.91
ATP7D	60.68	77.78	79.67	6.18	6.18	59.62	54.95	54.95	56.24
EDM	41.69	64.92	67.76	31.96	31.96	52.29	40.46	40.46	46.62
River flow 1	64.05	79.32	79.32	67.92	73.88	73.88	66.89	73.78	76.16
ENB	19.43	80.86	80.86	13.55	13.55	14.02	58.31	67.75	80.91
Jura	5.18	5.18	62.39	10.09	10.09	57.02	1.37	1.37	48.29
Online sales	9.93	44.71	51.29	5.52	5.52	62.17	10.33	10.33	70.36
Average	46.95	66.51	73.25	41.54	45.62	62.41	38.26	42.08	66.58

13.4 Other Solutions from Literature

We were able to find in literature only three instance selection algorithms for multi-label classification tasks:

- Kanj et al. [154] proposed a instance selection method, also based on the ENN algorithms, that aims to reduce noise in the dataset by removing outliers.
- Arnaiz-González et al. [155] recently proposed a method of adapting the local set concept, successfully used on single-label instance selection methods [22, 156], to multi-label datasets. It was used for adapting two single-label instance selection methods, LSSm and LSBo, to multi-label learning.
- Charte et al. [152] proposed a heuristic undersampling method for imbalanced multi-label datasets based on the canonical ENN method [12].

13.5 Conclusions

An instance selection method for multi-output regression problems has been proposed. In our opinion the results are excellent. The experimental validation of the method has shown that despite the large reduction of dataset size, in some cases by more than 50%,

the selected instances can be used to train a multi-output regressor, the performance of which is not worse and can even be better than having trained the regressor on the whole dataset.

The best results in terms of $rmse$ reduction were observed for the single regressor, which is obvious, as this method was internally used within the instance selection process. Also the best improvement was observed for 1-NN, as 1-NN is highly sensitive to noise and outliers, as it was already discussed in previous chapters.

A method of over-fitting prevention is also important and for that purpose we used a limited number of epochs with the possibility of generating up to three Pareto fronts that were then merged into one final front. Although the proposed method is based on the use of genetic algorithms, it is quite fast, because it uses all the mechanisms to increase the speed, which were presented in the previous chapters.

Part III
Instance Selection Embedded into Neural Networks

Chapter 14
Introduction to Neural Networks

Abstract In this chapter we outline how the neural network learns the data properties to be able to predict unknown similar data in classification and regression tasks. The kind of neural networks we will use in this part of the book will be Multilayer Perceptron and its modifications and extensions. We present the network structure and two learning algorithms: Rprop and VSS.

14.1 Introduction

Artificial neural networks are one of the most popular models applied to predictive analysis and have been used to a wide variety of tasks ([157, 158, 159, 160, 161]). Especially multilayer perceptrons (MLP) have been successively used in various applications, such as function approximation, classification, pattern recognition or signal and image processing. MLPs do not require any prior knowledge about input-output dependencies and are able to learn and build the data models based on training examples. For that reason they are popular and often considered easy-to-use tools, as there are many packages that implement neural networks.

Since the first successful neural network learning algorithm, called backpropagation, was developed [162, 163] the field of neural networks has been rapidly developing and a lot of neural network learning algorithms have been proposed in the literature. While working on instance selection with neural networks, we used mostly Rprop [166] and VSS [167] and we developed some extensions of these methods. As well Rprop as VSS make use of error surface properties.

In recent years the so called deep neural networks (networks with many hidden layers) are the most successful models in image recognition. In this work we will focus on neural networks used for classification and regression problems with the special emphasis on feature and instance selection and logical rule extraction.

The basic element of the neural network is a neuron. A neuron consists of two parts: the net function and the activation function. The activation function is also known as a transfer function. The net function determines how the input signals are combined inside the neuron The most commonly used net function and the only one considered in this work is given by the following formula:

$$u = \sum_{i=1}^{n} w_i x_i \tag{14.1}$$

The parameters $w-$ are called weights. The weight w_0 is called bias or threshold and its corresponding input signal x_0 always equals 1 and does not form a connection between two neurons as other weights do. The output of a neuron denoted by y is related to the output of the net function u by a transformation called activation (or transfer) function. Virtually any continuous non-linear and monotone function can be used as neural transfer function [164]. Moreover, if analytical gradient-based methods are used for network training, the functions must be differentiable. The most commonly used transfer functions for multilayer perceptron are hyperbolic tangent (Eq. 14.2-left) and logistic sigmoid (Eq. 14.2-right), where β is a coefficient determining the sigmoid slope.

$$y = \frac{1 - e^{-\beta u}}{1 + e^{-\beta u}} \qquad\qquad y = \frac{1}{1 + e^{-\beta u}} \tag{14.2}$$

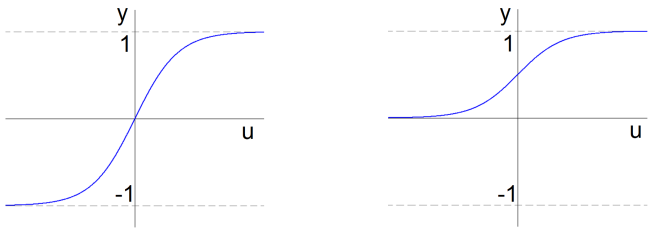


Fig. 14.1. Neural transfer functions: hiperbolic tangent (left) and logistic sigmoid (right).

14.2 Multilayer Perceptron (MLP)

In case of classification a single layer perceptron is able to classify only linearly separable data. For example, it is not able to solve the xor problem. Similarly in regression tasks it can reflect only very simple function approximations (linearly assuming linear transfer function).

A multilayer perceptron (MLP) is a neural network that contains the input and output layer of neurons and several (usually one or two) layers of neurons between input and output, so called hidden layers. However, for some special tasks, as image recognition also networks with more hidden layer are used. Moreover, there exist an optimal number of layers allowing for the best results in terms of prediction accuracy, which depends on the task the network performs. As a rule, the more complex task the higher number of layers or more hidden neurons provide the best prediction results.

The input layer is counted by some authors as a separate network layer and we will follow this convention. Thus in this work a three-layer network refers to a network of two layers of neurons with adaptable weights and one additional input layer of neurons that only distribute the input signals, as shown in the next figure. Some extensions of this architecture will be discussed in chapter 18.

In general the MLP networks used for classification and regression differ only by the output layer, which in case of classification consists of as many neurons with hyperbolic tangent or logistic sigmoid transfer function as the number of classes and for regression of only a single neuron with linear transfer function (although some exceptions are possible).

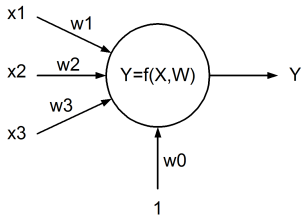


Fig. 14.2. Neuron model.

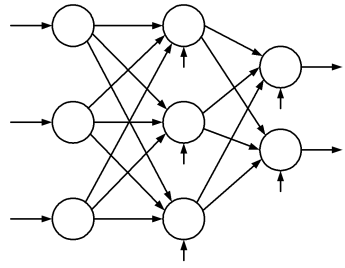


Fig. 14.3. MLP neural network.

14.3 Error Surface

To understand the learning process and the functioning of the network better, we can visualize the error function, which will be a hyper-surface in the weight space and for that reason it is called error surface [165]. As there are plenty of weights in the network and we can effectively see only in 3-dimensions, it is a good idea to choose two most characteristic directions $c1$ and $c2$ in the weight space and the vertical axis will present the error value in this space point, as in Fig. **??**.

A known method for finding the most characteristic directions, which contain the most variability and thus best shows the data properties is principal component analysis (PCA). We can obtain all network weights from each iteration of the learning algorithm and build a weight matrix. Then we perform PCA on that matrix to transform the original direction to the PCA component direction. It turns out that the first PCA direction usually explains about 80% of the total variability (the first eigenvalue is about 80% of all the values) and the second PCA component above 10%. Thus visualizing the error surface in the first and second PCA direction gives us a very good insight into the error surface properties.

As can be expected the shape of the error surface depends on two factors: the network architecture and the dataset properties. In the next figure we present typical error surfaces of MLP networks visualized in two PCA directions.

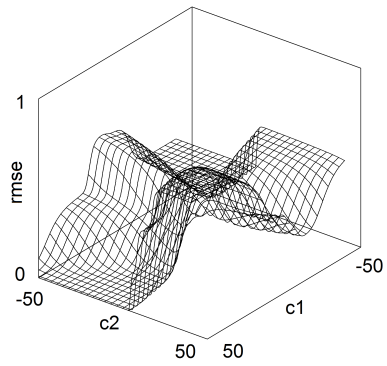


Fig. 14.4. Error Surface for Iris dataset. Horizontal axes: first (c1) and second (c2) PCA component.

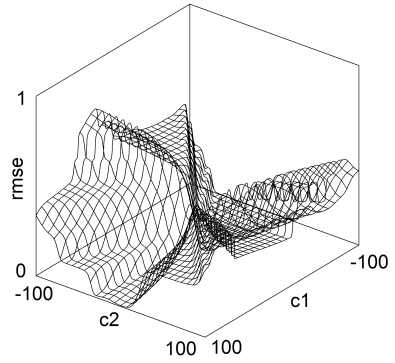


Fig. 14.5. Error Surface for Xor dataset. (source: our work [165])

From the PCA-based visualizations of the learning trajectories, we can observe that the change of each weight in current epoch is likely to be similar to the change in the previous epoch.

14.4 Neural Network Learning Algorithms

An MLP network performs a mapping from the input (feature) space to the output space. The aim of the network training is to obtain such weights (and such network structure if it is also modified by the training algorithm) that the mapping reflects the structure of the data and maximizes the classification accuracy or minimizes $rmse$ (or other error measure) as well for the training dataset as for the test dataset.

The network error, determined by the difference between the expected and actual network outputs, is a function of many parameters, such as the training dataset, network connection structure and weight values. In most cases the training data and network structure are not modified during the training, so the weight values are the only adjustable parameters. The network error function can be imagined as a multidimensional surface, with each weight defining one dimension. Thus, the training algorithms search for a minimum on the error surface.

The training set is given to the network inputs vector by vector, the network error is calculated based on the difference between expected and actual output(s) of the network and the weights are adjusted in order to minimize the error. The process is then repeated iteratively.

MLP training algorithms can be divided into several categories, such as analytical gradient-based, global optimization or search-based methods. Analytical-gradient based algorithms calculate the derivative of error function with respects to every weight and then change the weights in order to minimize the network error (by moving downwards on the error surface). Global optimization algorithms do not change the weights basing on the gradient direction but search for the minimum in much broader areas.

The training data frequently contains some noise and the noise should not be reflected in the mapping. If a network generalizes well then it achieves similar classification accuracy or $rmse$ in case of regression on a training set and on a test set. A test set contains vectors, which belong to the same data distribution, but which have never been used in the training process.

The neural network learning algorithms which at the same time can be used to train the network and to remove noise and redundant instances will be presented in the following chapters. However, as they are based on the general purpose algorithms, first we shortly present two of them: Rprop, which is based on backpropagation and the VSS algorithm developed by us.

14.4.1 Backpropagation and Rprop

The gradient-based MLP learning algorithms consist of two iteratively performed phases. In the first phase error gradients in each weight direction are calculated. In

the second phase the weights are updated taking into account the results of the first phase. Thus for several first-order (which means they calculate only the first derivatives) algorithms, as backpropagation, Rprop and Quickprop, the first phase is the same and only the second phase is different.

The sum-squared error function, which is minimized by backpropagation algorithm, can be written in the following way:

$$E = \frac{1}{2} \sum_{v} \sum_{j} (desired(v, j) - out(v, j))^2 \tag{14.3}$$

where $desired(v, j)$ is the desired signal and $out(v, j)$ is the actual signal of the j-th output neuron. The error is summed over all output neurons j and all vectors in the dataset v. The network weights are adjusted by a series of gradient descent updates. For sigmoid transfer function after some calculations that can be found literature, the equations that constitute the basic BP algorithm are obtained in the form presented below. We define

$$delta(k, n) = (desired(k, n) - out(n))out(n, k)(1 - out(n, k)) \tag{14.4}$$

as the $delta$ for the output layer, where n is the index of the layer. Then we backpropagate the deltas to earlier layers using

$$delta(k, n) = (\sum_{k} delta(n + 1, k)w(l, k, n + 1))out(n, k)(1 - out(n, k)) \tag{14.5}$$

where $w(l, k, n + 1)$ is the weight connecting the k-th neuron in the n-th layer with the l-th neuron in the n+1 layer. Then each weight update equation can be written as

$$w(k, l, n) = \eta(\sum_{v} delta(n - 1, j)out(n - 1, j) \tag{14.6}$$

To enhance the BP algorithm, variable learning rate and momentum (that is adding previous step to the current step with a certain weight to minimize the algorithm oscillation) can be used.

In Rprop, the second step in each weight direction does not depend on the gradient value but only on its direction. If the gradient in two successive epochs has the same direction (sign) the step in a given weight direction is increased (typically by 20%), other wise the direction of the step is changed and its value is decreased (typically by 50%). The assumption behind Rprop that each weight tends to change in similar way in two consecutive epochs proved to be more realistic than the assumption of backpropagation that each weight should be changed proportionally to its gradient compo-

nent. As a results Rprop in prevailing majority of cases outperform backpropagation. In our works we used mostly Rprop and the VSS algorithms.

14.4.2 Variable Step Search Algorithm (VSS)

We presented the final version of VSS (Variable Search Step Algorithm) in [168] and some improvements in [169]. VSS dynamically adjusts independently each weight during a rough minimization in each weight direction. VSS was designed taking the advantage of MLP error surface properties that its steepness in different directions varies ranks of orders, and the ravines in which the MLP learning trajectories lay are usually curves, slowly changing their directions [165, 168]. Basing on the properties we can expect that an optimal dw for the same weight in two successive training cycles will not differ much while dw for different weights in the same training cycle may differ ranks of order.

VSS changes a single network weight w by dw and checks if it caused the error to decrease. If yes, the change is kept and dw for this weight is increased, typically by 30-50%. If not then $dw = -0.5dw$ and the error surface is examined in the second point, from the other side of the current value of w, if the error decreases this weight value is kept, otherwise the weight w is not change in this iteration.

That gives VSS the advantage that it can better explore the error surface and thus requires much fewer iterations to train the network than Rprop, which changes all weights at once. Another advantage of VSS is that it can be used with any transfer functions, also noncontinuous and not differentiable and with any error function. To calculate the error in VSS we need to propagate the signals each time only via a small fragment of the network that is affected by the weight change, what makes the computational cost of one VSS iteration only a bit higher to that of one Rprop iteration.

Since only one weight is changed at a time, the signals do not have to be propagated through the entire network to calculate the error, but only through the fragment of the network in which the signals were different before and after the change. The remaining signals incoming to all neurons of hidden and output layers are remembered for each training vector in an array called "signal table". With VSS the signals must be propagated through the entire network only once at the beginning of the training thus filling the signal table. The dimension of the signal table is $N_V(N_H + N_O)$ where N_V is the number of vectors in the training set and N_H and N_O the number of hidden and output neurons. N_H denotes the sum of the neurons from all the hidden layers. After a single weight is changed, only the appropriate entries in signal table are updated. Also the error of each output neuron is stored in the error table and does not have to be calculated again if a weight of another output neuron is changed. The signal table reduces two types of calculations: summing the signals incoming to the neuron (since the sum of incoming signal is stored, it is enough to subtract the single old value of

one signal and add the new value), calculating the transfer function values. The error table reduces calculating the network error. It significantly shortens training times, especially for bigger networks. For a network structure 125-8-2 the tables reduce the training cost about 35 times, and for bigger networks even more, as shown in Fig. 14.6.

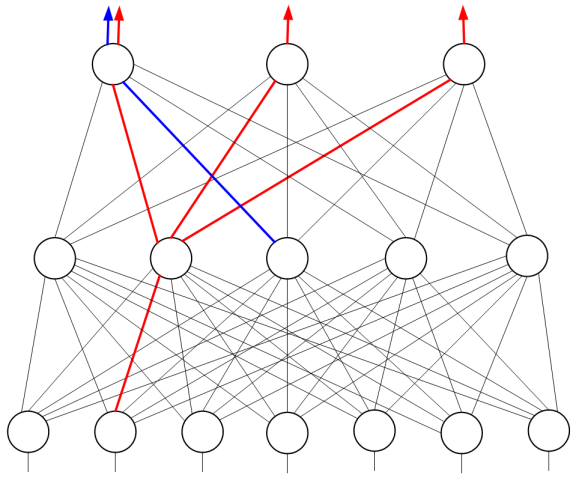


Fig. 14.6. Signals calculated after a single weight modification using signal table. In blue: when the output neuron weight is modified. In red: when the hidden neuron weight is modified.

In Fig. 14.6 signals that change if an output neuron weight is changed are shown in blue. Signals that change if a hidden neuron weight is changed are shown in red. The remaining signals are stored in the signal table and error table and thus there is no need to re-calculate them.

Another interesting option that can be quickly used for feature selection is initializing all hidden layer weights with zero values and setting the first guess dw of each weight change to a large value in the first training cycle. The larger dw causes that more features are eliminated from further training, because the results are better with the weight being zero than some big value. After the first training cycle all hidden weights that still equal zero are pruned and dw is again set to a smaller value. This is not the most accurate solution, but it is very fast.

Chapter 15
Noise Reduction in Neural Network Learning

Abstract In this chapter we discuss the methods of noise-resistant training of the MLP neural networks. Two groups of approaches are compared here - these based on noise removal by instance selection before network training and these based on modified robust error functions, which allow the network itself reduce the noise. The idea of the robust error functions is to limit the outlier influence on the network training by limiting network response to such instances.

15.1 Introduction

Instead of performing instance selection before the predictive model training, we can try to incorporate the instance selection into the model learning. One advantage of this approach is solving the problem of different decision borders of k-NN or similar algorithms (which are usually used inside instance selection) and the predictive model. Another advantage is the possibility of assessing during the model training how the selection influences the results and adjust the selection accordingly.

The drawback of this approach may be in some cases higher computational cost of the classification process than the joint cost of the prior instance selection followed by learning the classifier on the reduced set. This is especially evident for large datasets, where k-NN can be efficiently accelerated by methods like clustering and then performing the search for the nearest neighbors only within one cluster, KD-Tree [126] or Local Sensitive Hashing [127].

Yet another option is to join these two approaches, where the instance selection performed before the model learning removes only the most obvious cases to reduce the time of the training and the fine-tuned instance selection is incorporated in the model learning.

Unfortunately, the first and third option is not an universal approach, as different predictive models can perform the instance selection effectively in different situations.

Instances on which the trained neural network makes the highest errors are with high probability outliers. So the first approach to noise reduction may be to simply reject the instances for which the network error exceeds some threshold $maxError$. But this approach is associated with two problems: it is too late to reject the outliers after the network is trained, as they have already impacted the network mapping and there is a difficulty in setting an arbitrary $maxError$ value. For that reason to effectively deal with noise in the data special error measures and neural network learning algorithms have been designed.

In this chapter we will analyze how noise reduction can be implemented when the predictive model is an MLP neural network. Neural networks also help us find the redundant instances, especially in classification problems, but again to make it work efficiently dedicated solutions should be used, for example to enforce the network make much smaller errors on the redundant instances and to distinguished them in this way, but this will be discussed in chapter 17.

In chapter 16 we will discuss the approach to joint instance selection with similarity based and neural-network embedded methods and compare results of the methods described in this chapter and of the joint methods.

15.2 Noise Reduction With Error Function Modifications

Multilayer perceptron neural networks (MLP) are usually trained by minimizing $rmse$ on the training set. Since the $rmse$ measure uses the square of the error, the network is more strongly enforced to adjust to the instances, on which it makes the biggest errors. This is a very reasonable approach as long as the instances contain correct data. However, in practice, we are frequently not sure, which data points are correct and which contain wrong information. Typically the instances with the wrong information are outliers, this is they do no match their neighbors. Adjusting the neural network training to them is highly undesirable, and they should be rather removed from the training set than strongly influence the network learning, proportionally to the square of the error the networks makes on the outstanding instances. In this chapter we introduce the methods that will detect the outliers and prevent the neural network from representing them in the weight values.

Outliers and erroneous instances affect the network performance by leading to improper mapping from the input to the output space. The $rmse$ error function, can be considered as optimal only for clean training data or data containing at most values from zero-mean Gaussian distribution [4, 170]. Moreover, adding such noise to data is sometimes practiced in order to improve the network generalization. We can say that in this case we combat the noise with its own weapon. But this approach is used

in quite different scenario - when the points in the training dataset are situated too sparsely. In such cases, when the network maps exactly the correct but sparse points, the decision boundaries in classification or the function approximation in regression tasks can get far too complex to represent properly the data properties. However, the added noise, or rather added additional points, that are called "noise" is not the noise in the meaning of erroneous outliers. First, it has much smaller amplitude (smaller variability) and second it is generated to have the mentioned zero-mean Gaussian or similar distribution, while the distribution of the real noise is unpredictable.

Thus when the data contains lots of errors or outliers the $rmse$ measure becomes unreliable [171, 172]. To address this problem several methods, that use alternative error measures, were proposed [173, 174, 175, 176, 177]. The methods are called robust estimators and are mostly derived from robust statistical estimators. In these methods no instance selection or noise reduction is performed before the network training, but the training data is left in its original, potentially contaminated state, and it is the task of the neural network training process to deal with the noise. From that group we present in this chapter the following methods: ILMedS, LTA, LMLS, MAE, MIF, MedSum and various static an dynamic transformations of the error function, as trapezoid, exponential and others.

LTA, LMedS or bump error functions (like trapezoid or others) rely on the minimization of a saturated loss function or even a loss function that decreases as the distance between the two points grows beyond a certain limit. Indeed, this saturation or decrease ensures that outliers producing gross errors have a very limited impact on the estimation, as the gradient of the loss function in the saturated area is zero and in some cases can be even negative [178]. These functions are not continuous and non-differentiable, thus either some approximations of their derivatives in gradient-based learning or non-gradient based neural network learning algorithms are required. We use the second approach, training the networks with VSS (Variable Step Search algorithm) [167, 168], which is a non-gradient based learning algorithm, presented in chapter 14.

15.3 Static Robust Error Measures

The first approach we discuss to make the neural network training less sensitive to noise is to replace the $rmse$ error measure by other error measures based on the robust statistical methods.

As it was mentioned, the $rmse$ error measure is very sensitive to outliers, as it depends on the second power of the distance between the desired and the actual network output. This sensitivity is especially strong in regression problems, where the output neurons has linear transfer function and thus there is no limit on the values of its output signal. However, in classification tasks, this is also a problem, but not as severe,

because the output neurons implement hyperbolic tangent (or logistic sigmoid) functions, so the maximum difference between the actual and desired output raised to the second power equals 4 with hyperbolic tangent or 1 with logistic sigmoid.

In the domain of robust error measures the frequently used term is "residuals". The residuals r_i are defined as:

$$r_i = \sum_{v=1}^{q} |\bar{Y}(\mathbf{x}_i) - Y(\mathbf{x}_i)| \tag{15.1}$$

where i is the number of the instance and q is the number of the neural network outputs. So the residuals are just the sum of absolute values of the difference between the actual response of the network to the instance $\bar{Y}(\mathbf{x}_i)$ and the correct answer $Y(\mathbf{x}_i)$. We can write the performance function as:

$$E = \frac{1}{N} \sum_{i=1}^{N} \rho\left(r_i\right) \tag{15.2}$$

where $\rho(r_i)$ is a loss function [4], r_i is an error the network makes for the i-th instance (15.1), and N is the number of instances in the training set. The most frequently used loss function is a quadratic function:

$$\rho(r_i) = \frac{{r_i}^2}{2} \tag{15.3}$$

commonly used as the mean squared error (mse):

$$E_{mse} = \frac{1}{N} \sum_{i=1}^{N} {r_i}^2 \tag{15.4}$$

In was proposed to use the derivative of the loss function to assess the influence of the errors of the network responses on the network training [4, 171]:

$$\psi(r_i) = \frac{\partial \rho(r_i)}{\partial r_i} \tag{15.5}$$

If we consider this definition, for $rmse$ (or mse) performance function, the influence is linear, what means the greater errors more influence the network training.

To enable the neural network learning also in the presence of noise, several robust learning algorithms, which use modified error functions were proposed [179, 173, 174, 171, 180, 181]. In next section we describe some of the robust error measures, which reduce the impact of large residuals (which are usually caused by outlying data points) and in this way achieve the robustness to outliers. These measures can be divided into two groups: static an dynamic. Static measures remain in the same form during the whole neural network training process, while dynamic measures adjust their form

to the progress of network training. Moreover, there are not only appropriate error measures, but also the whole learning algorithms adjusted to high noise levels.

15.3.1 Least Trimmed Absolute Values (LTA)

The first group of error functions adjusted to high noise levels is based on quantile and trimmed performance measures [174, 177, 181]. A breakdown point of a robust statistical estimator is defined as the smallest percentage ϵ of contaminated data that can cause the estimator to take on aberrant values [4]. The most robust estimators have very high breakdown points; the best that can be expected is $\epsilon = 0.5$. However, as it will be shown in the experimental part, it comes with a certain cost; the network training quality is deteriorated in the absence of noise comparing to the standard $rmse$ error measure. For the $rmse$ $\epsilon = 0$, while for the least trimmed absolute value (LTA) and for the least median of squares (LMed) the breakdown point is close to $\epsilon = 0.5$.

The least trimmed absolute value estimator (LTA) is one of the classical high breakdown point robust estimators. It uses absolute values of residuals as L_1 norm, but the summation is replaced with a trimmed sum. For the general nonlinear regression model:

$$Y(\mathbf{x}_i) = f(x_i, \theta) + \epsilon_i, \quad i = 1, \ldots, n \tag{15.6}$$

where y_i is the output variable and $x_i = (x_{i1}, \ldots, x_{ik})$ is the independent input vector, θ is a parameter, and ϵ_i are the independent and identically distributed random errors. Using the LTA estimator a robust LTA error criterion [181] can be defined as:

$$E_{LTA} = \sum_{i=1}^{h} (|r|)_{i:n} \tag{15.7}$$

where $(|r|)_{1:n} \leq \cdots \leq (|r|)_{n:n}$ are ordered absolute network output residuals. Thus this error function (15.7) excludes from the training the instances causing largest errors, considering them outliers.

In [182] the upper limit h was based on the median of all absolute deviations from the median (MAD)[183]. The MAD is defined as:

$$\text{MAD}\ (r_i) = 1.483\ \text{median}|r_i - median(r_i)| \tag{15.8}$$

and the upper limit is in this case:

$$h = \|\{r_i : |r_i| < 3 * \text{MAD}(|r_i|), i = 1 \ldots n\}\| \tag{15.9}$$

15.3.2 Iterative Least Median of Squares (ILMedS)

ILMedS is a very interesting example, where it is not enough to employ a robust error measure, but the whole procedure of removing the instances exceeding the error threshold is repeated iteratively. It is derived from the least median of squares (LMedS) proposed in [184]. In the LMedS robust estimator not the sum of squared residuals is minimized, as in typical neural network learning, but their median:

$$E_{med} = \text{med } {r_i}^2 \tag{15.10}$$

An improvement to this measure was proposed in [174, 177], where after an initial training phase, the robust standard deviation (RSD)[184] is calculated as:

$$\sigma_r = 1.4826 * (1 + \frac{5}{(N - A)}) \sqrt{E^*_{med}} \tag{15.11}$$

where E^*_{med} is the best achieved LMedS error, where N is the number of instances and A is the number of attributes. Based on the RSD, all the training instances for which the network responded with residuals exceeding a threshold $r_i^2 \geq 2.5 * \sigma_r^2$ get removed from the training set. The entire procedure is repeated iteratively several times [174, 177]. In our experiments it was usually repeated 3 times.

15.3.3 Least Mean Log Squares (LMLS)

The error training function can be modified in many ways: in [171] Liano proposed a new LMLS (Least Mean Log Squares) error function based on so-called M-estimators, which should be optimal for the Cauchy distribution but performs well also for other long-tailed error distributions. This modification of the training algorithm was considered as referential in many works concerning robust learning methods [173, 174, 176]. The LMLS error is then defined as:

$$E_{LMLS} = \sum_{k=1}^{N} \sum_{i=1}^{m} \log(1 + \frac{1}{2} {r_{ki}}^2) \tag{15.12}$$

where r_{ki} is the error of i-th output for the k-th training set instance, N is the size of the training set and m is the number of network outputs.

A similar approach - the Hampel's hyperbolic tangent with additional scale estimator β, was used by Chen and Jain [179]. The scale estimator helped in determining the range of residuals believed to be outliers: all the residuals larger than β were partially excluded from the training procedure. Chuang and Su [173] proposed a similar

error performance function using the annealing scheme to decrease β with the training progress. In [172] a more sophisticated approach, using tau-estimators, was described.

15.3.4 MAE

This error formula can be also derived from robust M-estimators. As it was demonstrated in [174], the MAE criterion is probably the most effective of all constant error functions, when applied to training data with artificially introduced outliers. The mean absolute error is defined as:

$$E_{MAE} = \sum_{k=1}^{N} \sum_{i=1}^{m} |r_{ki}| \tag{15.13}$$

where r_{ki} is the error of i-th output for the k-th training set instance, N is the size of the training set and m is the number of network outputs.

15.3.5 Median Input Function (MIF)

The median input function was originally proposed in [180]. In this case the traditional summation of weighted neuron input signals is replaced with their median. The experiments showed that MIF improves tolerance to weights and neurons failures and also resistance to over-fitting. When the summation is replaced by more robust operation, such as median, the neuron output becomes less sensitive to the changes of its inputs. The MIF neuron output is defined as:

$$y_{out} = f(\text{med}\{w_i x_i\}_{i=1}^{N_{inp}}) \tag{15.14}$$

where $f(\cdot)$ denotes neuron transfer function (e.g. sigmoid or linear), x_j are neuron inputs, w_i is the i-th input weight and N_{inp} denotes input size. However, there are several problems concerning practical use of MIF. Calculating MIF output has higher computational cost than in the case of a simple sum. Moreover, the input function is not differentiable, so it cannot be simply trained with gradient-based methods. In [185] an approximated algorithm, based on the gradient for a simple sum, was presented. This approach seems to be very effective (also for non-differentiable error performance based on the median of residuals [177]). A combination of the error measure based on robust estimators was applied by El-Melegy in [175, 176].

15.3.6 MedSum

Our experience with MIF was not so optimistic and we decided to use it together with the standard summation function, which we called MedSum. Another problem with the MIF measure is that when it is used for regression tasks, the model built by the network can be non-continuous. This is why we decided to combine the median input with sum, defining new input function:

$$y_{out} = f(\delta \text{med}\{w_i x_i\}_{i=1}^{N} + (1-\delta)\sum_{i=1}^{N} w_i x_i) \tag{15.15}$$

where δ determines the median influence on neuron input function. The new MedSum (median plus sum) input function may help to overcome the problem of potential output discontinuity.

Also several other static robust error functions, as Charbonnier loss, Cauchy loss, Geman-McClure loss and Welsch loss were presented in literature [186].

15.4 Dynamic Robust Error Measures

The idea of dynamic error function is to smoothly reduce the outliers influence on the network training, as in the case of static functions, however, these functions change dynamically during the network learning process.

When the distance d between the actual and the desired network output is small the error value E grows with d according to the original error function. For example, if the original error function is quadratic, E is proportional to d^2. After d reaches the first critical value, depending on the implementation, E either still grows with d but slower, or E takes a constant value.

However, if d further grows and reaches the second critical value, E begins to decrease and finally reaches 0 and remains at 0 for any further growth of d. This corresponds to total elimination of this instance from the training process. Let us notice, that when E decreases its gradient is negative. Thus this approach may be problematic for gradient based MLP learning algorithms. However, it can be efficiently used with the VSS algorithm.

And the second difference. At the beginning of the training the network weights are random and high d value does not necessarily indicate an outlier. Thus, at the first epoch the standard error function has to be used with the critical points not used at all for classification tasks, where the output layer neuron response is limited between -1 and 1. In case of regression, the critical values can be set to very high absolute values, like -10 and +10. Then as the network training progresses, the weights gradually begin to reflect the data structure and the network makes the highest errors for the

outliers. To prevent the network adjust to the outliers, the critical points are gradually decreased up to their final values. This is the overall idea and implementation details are discussed in the next subsection.

15.4.1 Trapeziod, Exponential, Three-Parabolic and Triangular Error Functions

The exponential error function is presented in Fig. 15.1. It is basically the same function that was discussed in chapter 4, however this time it changes dynamically in a similar way as the trapezoid error function. The juxtaposition of three parabolas or triangular error functions are other possible solutions (see Fig. 15.1). The trapezoid error function is another example of that group [114] and it is explained in Algorithm 12 and Fig. 15.2

Algorithm 12 Neural network training with the trapezoid error function

for $epoch = 1 \ldots maxEpoch$ **do**
  $Error \leftarrow 0$
  $t1 \leftarrow 7.5$
  for $vector = 1 \ldots numVectors$ **do**
    $t1 \leftarrow t1/1.2$
    if $t1 < 3.0$ **then**
      $t1 \leftarrow 3.0$
    end if
    $t2 \leftarrow 1.5 \cdot t1$
    Calculate network output Y_{actual}
    $d \leftarrow |Y_{desired} - Y_{actual}|$
    if $d < t1$ **and** $d \leq t2$ **then**
      $d \leftarrow t1$
    end if
    if $d > t2$ **then**
      $d \leftarrow t1 - (d - t2)$
    end if
    if $d > 0$ **then**
      $Error \leftarrow Error + d$
    end if
  end for
  Modify network weights according to the training algorithm
end for

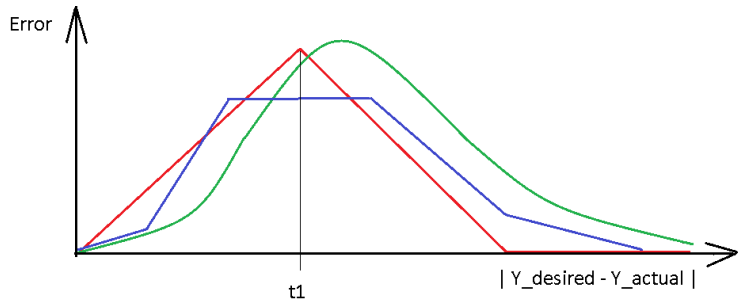


Fig. 15.1. Error functions for noise reduction: trapezoid (blue), triangular (red) and $parab$ - a juxtaposition of three parabolas (green).

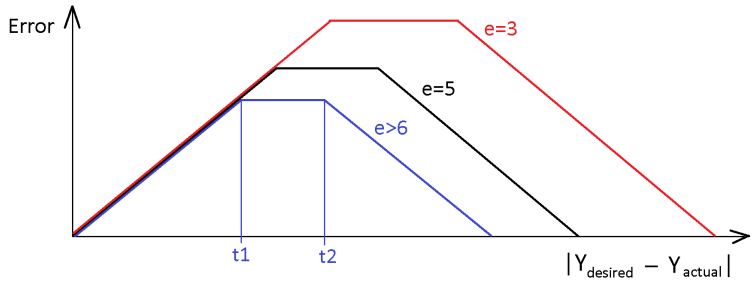


Fig. 15.2. Trapezoid error function (t1 and t2 asymptotically decrease), e is the epoch number of the VSS algorithm.

15.5 Experimental Evaluation and Conclusions

The experimental evaluation and conclusions will be presented in the next chapter on joining embedded and similarity based methods, where the methods presented in this chapter are compared with the joint methods. Moving the evaluation to the next chapter will allow to avoid presenting the same results twice.

Chapter 16
Joining Embedded and Similarity-based Instance Selection

Abstract Instance selection can be performed before the model learning. It can be also performed by the mechanisms embedded into neural network training. In this chapter we compare the two approaches and discuss the possibility and usefulness of performing them jointly, where the dataset is partially reduced before the neural network training and the mechanisms embedded into the neural network further reduce the dataset size.

16.1 Introduction

A question may appear: what is better, to perform the instance selection before the neural network training or let the neural network to perform the selection? In this chapter we will try to answer this question by experimental evaluation of several approaches. We can already say that although some methods work on average better than others, there is no single method (at least we have not found this) that works best for the whole spectrum of problems and thus the method should be adjusted to the data and our preferences. In this chapter we focus on noise reduction tasks, were using jointly the two methods can give the best results in terms of the neural network prediction accuracy. The data compression task embedded into neural networks is experimentally analyzed in the next chapter.

In the previous chapter we were discussing instance selection embedded into neural network training based on robust learning algorithms. In this chapter we discuss joint instance selection before and during neural network training and we compare the methods with those presented in the previous chapter. The evaluated method include those developed by us, as TEF, T-ENN and modified GAS and some robust error measures developed by other researchers. We present the experimental results to determine the quality of neural networks training with various instance selection algorithms for

different levels of noise and outliers added to the training sets. In the training data we added noise to outputs, inputs and to both: outputs and inputs and evaluated how each method performed for a given kind and amount of noise.

16.2 Experimental Evaluation

We performed the test in 10-fold cross-validation, as usually. For data reduction before neural network training we used the similarity-based instance selection with T-ENN and outlier detection with the modified k-NN Global Anomaly Score. We used a standard MLP network with one hidden layer and the number of hidden layer neurons set approximately to 50% of the number of attributes for each datasets. We used hyperbolic tangent transfer function in the hidden and output layer for classification and hyperbolic tangent transfer function in the hidden and linear transfer function in the output layer for regression. The number of output neurons for regression was obviously one and for classification was equal to the number of classes. The classification was considered correct if the signal generated be the output neuron associated with the class of the current instance was higher than the signals of all other output neurons (as is typically done).

The MLP network was trained with the VSS algorithm (see chapter 14) for between 10 and 18 iterations depending on the dataset size with different error functions described in the previous chapter ($rmse$, ILMedS, LTA, Medsum, TFE - trapezoid, $parab$ - a combination of three parabolas, MAE, LMLS) and a possible error weighting if k-NN Global Anomaly Score was used in the preceding step or discarding some of the instances if T-ENN was used in the preceding step. The network error on the test set was evaluated using $rmse$ error measure for regression tasks and classification accuracy for classification tasks, even if a different error measure was used in the training. That allowed to directly compare the results obtained with different methods.

Since the purpose of the experiments was not to select the optimal neural network architecture and number of iterations, but to evaluate the noise-robust methods and their combinations with the similarity-based instance selection algorithms, we did not perform any optimization of the two parameters. Moreover, we were rather interested here in comparing the performance of different noise reduction methods than in maximizing the network performance with all possible means.

We present the results of experiments using various amount of noise introduced to the original dataset. The noise had Gaussian distribution amplitude with zero mean and standard deviation of d. Each input value of each instance was modified with a probability p by adding a random value from the mentioned Gaussian distribution. The same was done for the outputs in regression tasks. For the symbols given in Tables 16.2 and 16.3 and the corresponding Fig. 16.1 and 16.2, the random noise with the

parameters shown in Table 16.1 was added. For classification tasks, with probability p the class of each instance was substituted with a class of a random instance.

Table 16.1. The parameters of noise added to the datasets. input/output d - amplitude of noise added to inputs/output, input p - probability of adding noise to each input/output.

	0	i1	i2	i3	i4	i5	o1	o2	o3	o4	o5	io1	io2	io3	io4	io5
input d	0.0	0.5	0.8	1.5	2.5	4.0	0.0	0.0	0.0	0.0	0.0	0.5	0.8	1.5	2.5	4.0
input p	0.0	0.1	0.2	0.3	0.4	0.5	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.3	0.4	0.5
output d	0.0	0.0	0.0	0.0	0.0	0.0	0.5	0.8	1.5	2.5	4.0	0.5	0.8	1.5	2.5	4.0
output p	0.0	0.0	0.0	0.0	0.0	0.0	0.1	0.2	0.3	0.4	0.5	0.1	0.2	0.3	0.4	0.5

Some of the experimental results were partially published in our previous papers, especially the results with MAE, LMLS, and MedSum were published in our work [97] and we will not present the detail results of these methods here, because there are already 15 methods presented in Fig. 16.1 and comparing more methods would be impractical. To give the reader an idea how the three methods perform: their lines in Fig. 16.1 would be situated between ILMeds and LTA, rather closed to ILMeds.

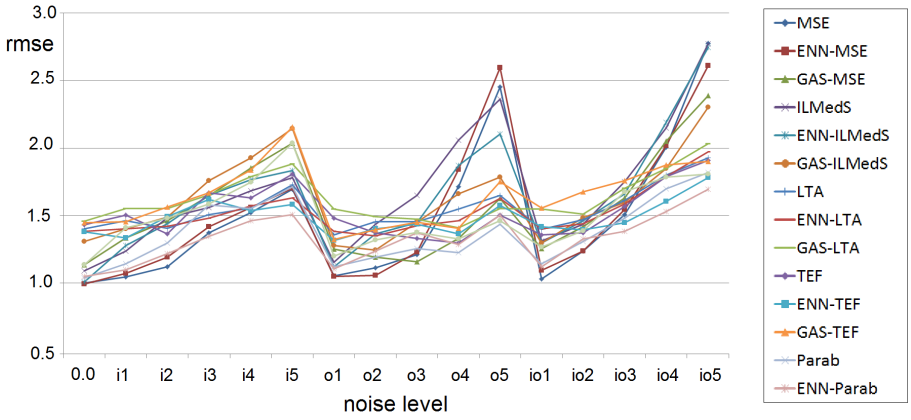


Fig. 16.1. Experimental results for regression datasets with added noise. Horizontal axis: amount and location of added noise - see Table. 16.1. Vertical axis: average $rmse$ for all tested regression datasets.

As it can be seen the $parab$ error measure gives very good results, but this error measure was especially designed to give good results in these tests. In order to achieve this for low errors it becomes the standard quadratic function, the same that is used by

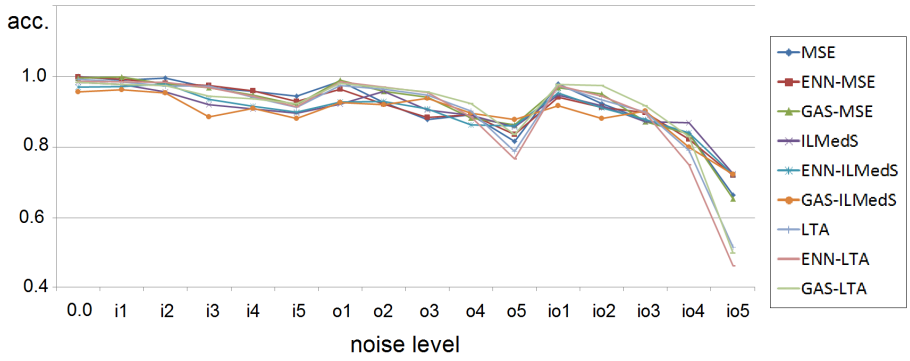


Fig. 16.2. Experimental results for classification datasets with added noise. Horizontal axis: amount and location of added noise - see Table. 16.1. Vertical axis: average classification accuracy for all tested classification datasets.

Table 16.2. Average relative $rmse$ in 10-fold cross-validation for the regression datasets with various amount of noise added.

	0	i1	i2	i3	i4	i5	o1	o2	o3	o4	o5	io1	io2	io3	io4	io5
MSE	1.00	1.05	1.12	1.37	1.52	1.70	1.06	1.11	1.21	1.71	2.45	1.03	1.24	1.51	2.01	2.78
ENN-MSE	1.00	1.07	1.19	1.42	1.56	1.70	1.05	1.06	1.23	1.84	2.59	1.10	1.24	1.55	2.02	2.61
GAS-MSE	1.14	1.34	1.46	1.65	1.85	2.04	1.25	1.19	1.16	1.33	1.63	1.26	1.42	1.63	2.05	2.39
ILMedS	1.09	1.24	1.48	1.56	1.68	1.78	1.16	1.44	1.65	2.06	2.36	1.31	1.44	1.75	2.15	2.77
ENN-ILMedS	1.01	1.28	1.44	1.65	1.77	1.84	1.12	1.37	1.45	1.87	2.10	1.30	1.46	1.66	2.19	2.75
GAS-ILMedS	1.31	1.40	1.49	1.76	1.93	2.14	1.28	1.25	1.45	1.66	1.79	1.30	1.49	1.60	1.85	2.30
LTA	1.40	1.46	1.41	1.51	1.56	1.73	1.36	1.46	1.46	1.55	1.65	1.40	1.47	1.61	1.80	1.93
ENN-LTA	1.38	1.40	1.42	1.48	1.57	1.63	1.39	1.35	1.42	1.46	1.63	1.40	1.44	1.59	1.79	1.97
GAS-LTA	1.46	1.55	1.55	1.65	1.79	1.88	1.55	1.49	1.47	1.41	1.55	1.55	1.51	1.70	1.85	2.03
TEF	1.43	1.50	1.36	1.67	1.63	1.81	1.48	1.37	1.33	1.30	1.50	1.36	1.37	1.57	1.79	1.91
ENN-TEF	1.38	1.34	1.49	1.62	1.54	1.58	1.32	1.40	1.44	1.36	1.57	1.42	1.39	1.45	1.61	1.78
GAS-TEF	1.45	1.45	1.56	1.67	1.84	2.16	1.32	1.40	1.45	1.41	1.75	1.56	1.68	1.76	1.87	1.90
Parab	1.04	1.14	1.30	1.58	1.55	1.71	1.12	1.19	1.26	1.23	1.44	1.14	1.31	1.48	1.70	1.82
ENN-Parab	1.05	1.10	1.22	1.35	1.46	1.51	1.11	1.24	1.37	1.29	1.50	1.12	1.33	1.38	1.53	1.70
GAS-Parab	1.14	1.41	1.48	1.59	1.75	2.04	1.20	1.32	1.37	1.33	1.46	1.27	1.39	1.68	1.78	1.81

$rmse$ for small error values. For that reason it performs better in the experiments than the trapezoid error measure, although both of this measures follow the same scheme: they initially increase with the error and then decrease. But this does not mean that the $parab$ error measure is better in practical applications than the trapezoid measure, because in practical applications not always the preferred error measure is $rmse$.

Table 16.3. Average relative classification accuracy in 10-fold cross-validation for the classification datasets with various amount of noise added.

	0	i1	i2	i3	i4	i5	o1	o2	o3	o4	o5	io1	io2	io3	io4	io5
MSE	1.00	0.99	1.00	0.97	0.96	0.95	0.99	0.93	0.88	0.89	0.82	0.98	0.92	0.88	0.84	0.67
ENN-MSE	1.00	0.99	0.98	0.98	0.96	0.93	0.96	0.92	0.88	0.89	0.84	0.94	0.91	0.90	0.82	0.72
GAS-MSE	1.00	1.00	0.98	0.97	0.95	0.92	0.99	0.96	0.94	0.88	0.86	0.97	0.95	0.87	0.84	0.66
ILMedS	0.98	0.98	0.96	0.92	0.91	0.90	0.92	0.96	0.91	0.89	0.86	0.95	0.92	0.87	0.87	0.73
ENN-ILMedS	0.97	0.97	0.98	0.94	0.92	0.90	0.93	0.93	0.91	0.86	0.86	0.95	0.91	0.88	0.84	0.72
GAS-ILMedS	0.96	0.96	0.95	0.89	0.91	0.88	0.93	0.92	0.94	0.90	0.88	0.92	0.88	0.90	0.80	0.72
LTA	0.99	0.99	0.97	0.97	0.94	0.91	0.98	0.96	0.95	0.90	0.79	0.98	0.93	0.90	0.79	0.52
ENN-LTA	0.99	0.99	0.98	0.97	0.95	0.92	0.99	0.97	0.96	0.89	0.77	0.98	0.95	0.90	0.75	0.47
GAS-LTA	0.98	0.98	0.98	0.95	0.94	0.92	0.98	0.97	0.96	0.92	0.84	0.98	0.98	0.92	0.83	0.50

16.3 Conclusions

The experimental evaluations clearly show that there is no a single best noise reduction methods that covers the whole spectrum of problems. Instead we suggest to adjust the noise reduction method to the amount of noise in the data. The standard MLP training with the $rmse$ error measure works best for high quality data, that contains very little noise. There may obviously arise the question how can we now the noise level in our data. Frequently we can't know this ahead and therefore it is suggested in such situations to assess the noise level.

One way to do it is by training the network with the standard $rmse$ error measure. In our experiments if for standardized outputs, $rmse$ on the test set was below 0.15 then the amount of noise was low and likely $rmse$ would be the best measure. If it is above 0.5 than the robust error measures should be better. Also based on some statistics we can predict, which methods will achieve best results in accuracy or $rmse$, as was discussed in the part 1, chapter 1 of the book, based on the performance of 1-NN algorithm we can assess the amount of noise in the data and make an informed guess of the best method.

When the results for both types of problems, regression and classification are considered, combination of ENN and LTA algorithms shows the best performance. In most regression datasets the TEF and $parab$ methods performed especially well, in some cases allowing to obtain $rmse$ on the test set up to 40% lower than any other method. The regular $rmse$ error measure, even for not contaminated data, in most classification tasks was outperformed by robust methods. Also joining the modified k-NN GAS algorithm with the MedSum error measure proved to work well for regression tasks, comparatively to LTA. This can be explained by the fact that when a more stable operation, such as median is added to the summation in the neuron function, the neuron output becomes less sensitive to perturbances in the data.

For classification tasks, if the noise was added to output or to both output and input the best methods were combinations of GAS and LTA, except for the highest level of noise, where a combination of GAS and ILMedS or ENN and ILMedS performed better. Another interesting finding is that the modified k-NN GAS usually allowed for stronger noise reduction than ENN. However, for classification tasks the differences between particular methods of noise reduction were smaller than for regression tasks. That is also because, the robust learning algorithms (LTA and ILMedS) were designed to work with regression tasks and in classification, where the errors are limited by the hyperbolic tangent transfer function they cannot show their full effectiveness.

The ENN and GAS methods combined with different error measures usually keep their general direction: if the error measure was stable with increasing noise, the hybrid version of the method is also stable and improves the results a little. When the error grows rapidly for the basic algorithm, when the noise increases, it grows also for the modified method. In this cases the ENN or GAS reduced the error significantly, although the results were still much worse than with using a robust learning algorithm (robust error measure).

Chapter 17
Joint Feature and Instance Selection from Neural Networks

Abstract There are two basic ways to reduce the training data size: feature selection and instance selection. Using these techniques appropriately can also improve the model prediction. When the model is a neural network, we have plenty of options to reduce the data size (feature selection or instance selection first, features or instances selection before the network training or during the network training either sequentially or simultaneously). In this chapter we discuss the options and present our recommendations.

17.1 Introduction

In embedded methods, feature selection is an integrated part of the predictive model learning. Thus, at the same time, the model is trained and feature selection is performed. In terms of simplicity this is a very good solution. However, if the dataset is very large, the computational cost of that approach can still be very high and then using feature filters can be a more practical approach. For example, decision trees perform forward selection. Linear regression does not directly perform feature selection, but allows us reject those features for which the smallest weights are assigned. Similarly with the neural network - one can reject these features, for which the sum of weights of all neurons of the first hidden layer is the smallest. In addition, in embedded methods, we can use regularization by using an additional penalty term for too complex models in the objective function or more advanced regularization algorithms.

Although feature selection and logical rule extraction are closely related topics, we decided to split them into two separate chapters, to improve the organization of the book.

Rule extraction methods perform also feature selection as a by-product. This can be especially clear that when a given feature is eliminated in the selection process -

then it will not be included in any logical rule. However, the feature selection methods do not provide logical rules, although the features that obtain higher weights in feature filters are more likely to be included in the rules, which cover more examples. In this chapter, we will first review the approaches to feature selection from neural networks. We will focus on the decompositional methods, where the feature selection or feature weighting is performed by the analysis of the network structure and connections.

Then we will experimentally evaluate some possible combinations of feature and instance selection in neural network training. We will start from evaluating different feature rankings to find the best one for our purposes. Then we will evaluate the similarity-based instance selection methods and choose the ones, which best perform with our data and neural network. Then we will analyze the mechanisms of feature and instance selection embedded into neural networks and finally we will draw some conclusions. As there are thousands of possible combinations, we will not even try to test all of them, but rather focus on the groups of solutions and discuss their strong and weak points and the possibility of using different methods together.

17.2 Feature Selection Embedded into Neural Network Learning

Feature selection with feature filters before the model training and with feature wrappers was presented in chapter 6. Thus in this chapter we will focus on feature selection embedded into the neural network training.

Feature selection with neural networks can be done in several ways. The two basic approaches are by the analysis of weights, including pruning methods and by input data perturbances [187, 188, 189, 190]. In perturbance analysis we replace the values of particular feature with random values or we temporarily delete it in the test vectors and see how this influences the network accuracy [191]. In weight analysis we assume that the less important features will generate smaller absolute values of weights and we can reject the features with the lower weighted sum of weights r_i. The weights can be also enforced to small values with a regularization term. The weight analysis was used in our experiments.

More complex methods can also consider the derivatives or the output neuron weights. Due to the non-linear transfer functions the results depend on the actual position on the transfer function and in classification task at the end of the training the position is predominantly in the saturated area, so there must be some more effort put into constructing an efficient solution. We used the following feature ranking measure:

$$r_i = \sum_{h=1}^{H} \frac{|w_{in}|}{\sum_{f=1}^{F} |w_{fn}|} \tag{17.1}$$

where r_i is the predictive power of the i-th feature, H is the number of hidden layer neurons, F is the numbers of features, w_{in} is the weight connecting the h-th hidden neuron with the i-th feature and w_{fn} $(= 0...F)$ is the f-th weight of the h-th hidden neuron.

17.3 Other Methods from Literature

In [192] a feature weighting method for classification tasks by extracting relevant information from a trained neural network was proposed. This method weighted the attributes based on strengths of related weights in the neural network, assuming, that stronger connections from the input assigned to that attribute to output mean that the attribute is more important. After the training, the attribute weighting was extracted from the neural network in the following way:

$$W_i = \sum_{j=1}^{H} \sum_{k=1}^{O} |V_{i,j} \times V_{j,k}| \tag{17.2}$$

where W_i is the feature weight for input node i, $V_{i,j}$ is the link strength from input node i to hidden node j, and $V_{j,k}$ is the weight from hidden node j to output node k, H is the number of hidden nodes and O the number of output nodes. The double summation covers all possible paths from input node representing a given features to all output nodes.

In [193] an instance-wise feature selection method called INVASE was proposed. The method consists of 3 neural networks, a selector network, a predictor network and a baseline network used to train the selector network with the actor-critic methodology. The advantage of the INVASE is the flexibly of discovering different feature subsets for each instance.

In [194] the authors presented a feature selection and an outlier detection method. If the data is very noisy, first the outlier detection and removal is used. In order to accomplish this, the centroids of each class are computed and the instances laying very far from them are considered outliers. The second step is feature selection performed by a genetic algorithm, where the features are encoded in the chromosome and the neural network is used as the evaluator model.

A method called DeepLIFT [195] transposed the output of the neural network to a reference input in order to compute the contribution of each input variable.

In [196] Shapley values were used to compute the variable importance, and locally linear models to explain the linear dependency for each sample.

17.4 Data Reduction with Boundary Vectors in Neural Networks

In this section we focus on instance selection to reduce (condense) the data size in classification tasks using the concept of "boundary vectors". However this step should be performed after noise removal, because always noise should be removed first from the data and then data size reduction can be performed (the noise removal embedded into neural networks was discussed in chapter 15).

In classification problems we need to determine the class boundaries and to obtain this we need only the instances, which are situated close to the boundaries. So the instances that are far from the decision boundaries and surrounded by other instances of the same class can be removed, as they do not bring any useful information to the classification problem. This idea of boundary vectors known from support vector machines can be also applied to neural networks [197, 198]. However, there is still problem how to identify those instances.

While removing irrelevant examples as those, on which the neural network makes the smallest errors, the examples that get removed are those far from the decision border, so those that are not necessary to determine the decision border and thus the decision border remains intact.

For classification tasks we use usually an MLP neural network with hyperbolic tangent or logistic sigmoid transfer function with as many neurons in the output layer as the number of classes. When an instance is processed by the trained neural network, the output neuron associated with this instance class gives signal = 1 and all the other output vectors associated with different classes produce the output signal = -1 for hyperbolic tangent transfer function or 0 for logistic sigmoid. The error for a single vector $\mathbf{x}_i$ used for instance selection is given by the following formula:

$$Error(\mathbf{x}_i) = \sum_{i=1}^{n_c} (y_{ai} - y_{ei})^2 \tag{17.3}$$

Where n_c is the number of classes, which equals the number of output layer neurons, y_{ai} is the actual value of i-th output neuron signal and y_{ei} is the expected value of i-th output neuron signal (which is 1 if the current instance class is represented by the i-th output neuron and -1 or 0 otherwise). We assume that a vector is classified correctly if the neuron associated with its class gives a higher signal than any other output neuron. If an instance of the training set is classified incorrectly by a trained neural network, the error that the network gives as a response to that instance is high ($Error(\mathbf{x}_i) > maxError$). On the other hand if an instance is classified correctly and is situated far from a classification boundary then it is located on the saturated part of the transfer function. In this case, the network error for that instance will be very low ($Error(\mathbf{x}_i) < minError$). Thus this instance can be removed from the training set $\mathbf{T}$. In theory that can be done after the network training. In this aspect it differs from noise reduction, where the reduction must be performed during the train-

ing to prevent the noisy instances from influencing the network weights. The instances located in the middle of their classes do not influence network weighs, as the error the networks makes on them is so small, that it has no practical impact on the weight changes. Only the instances close to decision boundaries support the network learning process.

There still remains a question how to identify these instances. The instance removal can be done after the training only in theory as it was written, but in practice the neural network frequently works too well for this approach. Thus the removal should be done earlier, by gradually removing the instances on which the network already generates very small errors. As the experiments showed, when the neural network is trained in a typical way, then frequently the total error can get very low. It is so low that it can take almost zero values as well for the "inner" instances as for the boundary instances. The pseudo-code for the embedded instance selection for data size reduction is shown in Algorithm 13.

Algorithm 13 Instance selection embedded in neural network training

Require: $\mathbf{T}, minError$
  $n \leftarrow |\mathbf{T}|$;
  for $i = 1 \ldots I$ **do**
    train the network on $\mathbf{T}$
    for $n = 1 \ldots N$ **do**
      if $Error(\mathbf{x}_n) < minError$ **then**
        $\mathbf{T} \leftarrow \mathbf{T} \setminus \mathbf{x}_n$
        $N \leftarrow N - 1$
      end if
    end for
    adjust $minError$
  end for

The network is trained for I iterations with some training algorithm. After each iteration i the error the network makes on each of the N instances is recorded and if it is below $minError$, the instance is removed. Then $minError$ is adjusted by decreasing it, because as the training progresses the network tends to make smaller error on each instance x_n, which is not an outlier. As the $minError$ depends more on the neuron weight values, a better solution than using a constant value is to use a relative value in relation to the error the network makes on other examples. We use for $minError$ some percentage of the average error values of all correctly classified instances of a given class.

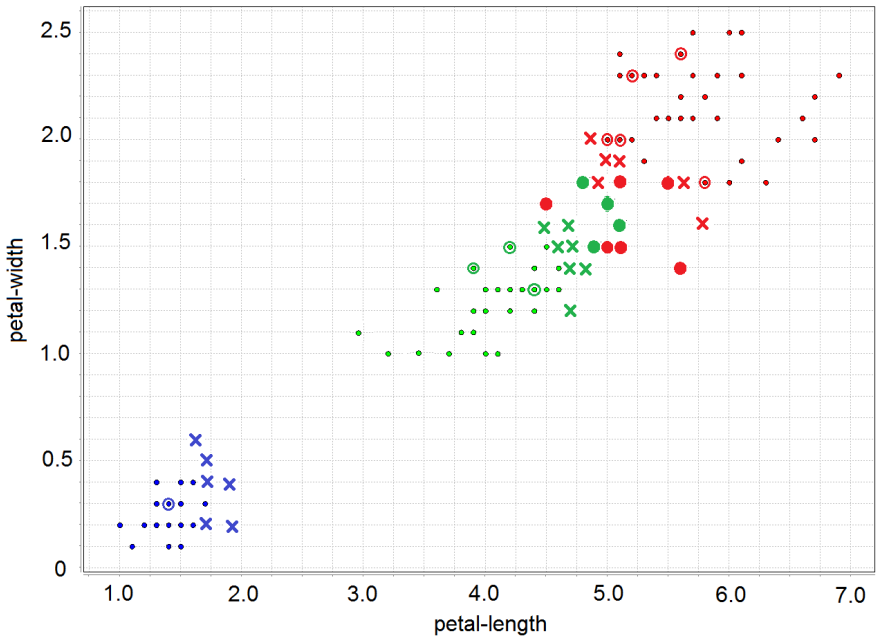


Fig. 17.1. The instances of Iris dataset selected by CNN (dot inside circle), by neural network (cross) as those with low error values and by both (big solid circle). As it can be seen the neural network selected instances, which are closed to the borders of the classes. Color denotes class. (figure based on our work [198])

17.5 Joint Feature and Instance Selection Embedded into Network Learning

To determine the optimal order of joint feature and instance selection with neural networks, we conducted experiments trying feature selection first, instance selection first and simultaneous feature and instance selection. As in the case of the selection prior to network learning, the results confirmed that the best option in most typical cases is to perform feature selection first and then instance selection. Thus the network training consists of three parts: 1. standard network training, 2. removal of irrelevant features, 3. removal of irrelevant instances. After the second step the training can either continue or it can be restarted from random weights. Better results were obtained with the restart.

We observed in the experiments that instance selection embedded in neural networks worked well, but feature selection was in some cases as good as done with feature filters and in other cases less effective. For that reason we added another op-

tion for feature selection. First we build a simple neural network with three hidden units and train this network separately on each feature. We use early stopping, so that we can measure the classification accuracy on a training set, without the need of cross-validation. Then we sort the features by the classification accuracy. Then we add the features to the reduced dataset starting from the most informative one. However, before we add the next one, we calculate its correlation with all the already added features. If the correlation with at least one of them is higher than the threshold, we reject this feature. This appeared to be a simple and yet quite accurate option.

17.6 Experimental Evaluation

We conducted the experiments to evaluate particular methods in terms of classification accuracy and data compression and determine the Pareto front for each method in the compression-accuracy coordinates, as shown in Fig. 17.2.

Table 17.1. Average values over the 10 datasets of classification accuracy of neural networks (F100-A, F60-A, F30-A) and number of selected instances (F100-I, F60-I, F30-I) for respectively 100%, 60% and 30% of features with three data selection methods. The real numbers of features were the nearest integers to these percentages. (based on our work [198])

Method	IS	F100-A	F100-I	F60-A	F60-I	F30-A	F30-I
FS: Inf. Gain	no selection	92.74	100	92.12	100	91.02	100
IS: ENN+CNN	m=8, k=9	93.01	35.22	92.30	29.11	91.03	27.81
with variable	m=7, k=9	92.65	18.91	91.58	16.15	88.76	15.11
m in k-NN	m=5, k=9	87.44	7.11	87.11	5.89	87.72	6.41
IS: ENN+IB3	ENN+IB3	87.15	3.85	86.88	3.98	87.01	4.95
FS in separate	no selection	92.74	100	92.90	100	91.45	100
network	minE=0.03	93.02	38.45	92.30	32.98	91.23	36.40
IS embedded	minE=0.1	92.64	19.98	91.91	22.12	88.76	26.78
into NN	minE=0.3	89.05	7.11	88.23	5.89	88.25	10.23
FS embedded	no selection	92.74	100	92.12	100	88.34	100
into NN	minE=0.03	93.02	8.45	92.15	33.15	88.94	38.14
IS embedded	minE=0.1	92.61	19.98	91.05	24.98	87.02	14.18
into NN	minE=0.3	89.05	7.11	87.91	9.15	86.15	10.78

The neural networks with one hidden layer was trained using the Rprop algorithm. The numbers of neurons in the hidden layer was equal to the geometric mean of the number of inputs and number of classes. We performed the experiments on 10 clas-

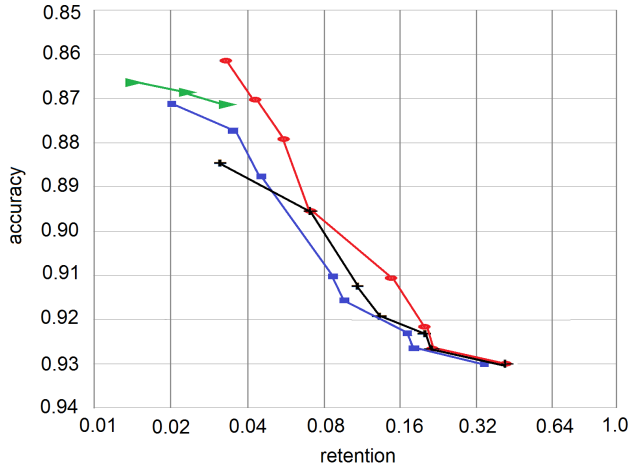


Fig. 17.2. Retention - accuracy plot (note: the retention axis is in logarithmic scale). A Pareto line is shown separately for each method. Blue rectangle = FS: Inf. Gain, IS: Mod. ENN+CNN. Black cross = FS in sep. network, IS embedded. Red ellipse = FS and IS embedded. Green triangle: FS: Inf. Gain, IS: ENN+IB3.

sification datasets from the Keel Repository [8] (Ionosphere, Image Segmentation, Magic, Thyroid, Page-blocks, Shuttle, Sonar, Satellite Image, Penbased, Ring). As always, all the experiments were performed in 10-fold cross-validation and repeated 10 times. We used similarity-based and the described in this chapter embedded into neural network instance and feature selection methods.

17.7 Conclusions

For the biggest datasets in classification tasks we were able to remove over 90% of instances without noticeably accuracy loss, but for smaller datasets the reduction was much weaker.

For most datasets, the most effective data selection is performed by feature selection followed by instance selection. This is true as well for the selection prior to network training as for embedded into the neural network. That problem was already discussed in details in chapter 6, section 6.4, so the results should not be surprising. As it was discussed it is not always best to perform feature selection first, but to perform noise reduction before data condensation and in most datasets it happens that

more noise is associated with features than with instances, but it does not have to be so every time.

The Pareto front for the selection with information gain and ENN+IB3 and then modified ENN+CNN was situated closest to the right lower corner, so this method worked here best. However, each of the methods have some strengths and weaknesses.

Feature ranking obtained by learning a simple neural network with single feature datasets and with removal of highly correlated features worked very well. The standard feature rankings, as information gain, were on the second place, while feature selection by neural network weight analysis on the third place. However, the last method can be further enhanced by considering the data flow through the entire network, not only the input to hidden weights and thus may produce better results.

Embedding noise reduction into the neural network learning process gives usually very good results. That can be attributed to the shape of decision boundaries, where the k-NN algorithm have the tendencies to smooth the edges (see Fig. 2.2 in chapter 15).

Instance selection as noise removal worked quite well in each case. There was however one problem with instance selection as data compression. Both DROP-3 and the instance selection based on the network error overcomes the shortage of CNN that it works in a random order, as they both preserve more of the instances situated close to class boundaries. However, both of the methods rely on the distance to the opposite class measured either directly (CNN, IB3 and the DROP family) or by the instance location reflected by error produced by the hyperbolic tangent function transformation. But both of the approaches do not consider the fact, that the distance between opposite class instances may be different in different areas of the input space and thus sometimes they tend to remove rather the instances closest to the boundary, even if they are behind the "first front" of instances than the instances that are further, but in the "first front" and thus are needed to preserve the boundary. That is considered by other instance selection methods, which examine the classes of neighbor instances of Voronoi cells, but in spite of that they do not perform better. Finding an effective solution to this problem is still open.

It is likely that the results can be further improved if other instance selection algorithm, as IB3 or DROP-3 are applied or m of k neighbors in the same class are required in the weighted k-NN (where m is a parameter). The later was recently partially addressed by an instance selection method that optimizes the k in k-NN individually for each instance [103]. In the same work the authors allowed for selecting one instance more than once, what makes sense for some learning models reducing their training time, by adding an additional column with the number of how many times a given instance is selected and just multiplying the result for this instance by this number.

Chapter 18
Special Neural Networks for Data Selection and Rule Extraction

Abstract Logical rule extracted from neural networks explain the decisions made by the network and the properties of the data discovered by the network. The decompositional methods of logical rule extraction analyze the connections between neurons in a neural network. Because the system of connections among neurons is very complex, including the complex interpretation of the neural transfer functions, the obtained rules tend to be complex and not very accurate. In this chapter we address these shortcomings, by constructing an incremental network with simple connection structure and with dedicated hidden neurons for each class. This removes the difficult-to-solve interferences between various logical rules and makes the rule extraction and data selection much easier.

18.1 Introduction

Neural networks belong to the best of the prediction models. However, the problem connected with them is that they are usually treated as black boxes that map some input variables to some output variables, but we do not know why they make the decisions they make. That makes some problems with implementing neural-network based predictors to the tasks where the user needs justification why particular decision was made. Many users do not want to trust a prediction model that they do not understand and consequently will not use it for critical and important tasks.

Logical rule extraction from neural networks is the process that results in presenting us the knowledge discovered in the data in a form of simple logical rules. This has two advantages: 1. the user will trust the network results, 2. the user can enhance their knowledge about the process by learning some new dependencies discovered by the neural network.

Especially extracting rules understandable for humans for the nonlinear regression problems is a complex and challenging task, which is always associated with a trade-off between the rule complexity and thus the ability of humans to understand them and the rule accuracy. Moreover, there is not strict consensus how the rule understandability should be measured, therefore it is frequently difficult to tell that one method of rule extraction is superior to the other. That is also true for logical rule extraction for classification tasks.

The research on logical rule extraction from neural networks began on a wide scale in 1990-ies and continues till these days. Thus a lot of algorithms were proposed. We will shortly present some of the most characteristic ones, mainly from the decompositional group but also some pedagogical and hybrid methods.

Also instance selection is connected with rule extraction, as the instances that are not covered by any rule generated from the network can frequently be considered noise and thus can be removed. Moreover, the neural network is frequently able to reduce the noise, thus better reflecting the problem properties than the raw dataset describing the problem.

The simplest solutions are the best if they provide the same results as more complex ones. The simplest rules are those that are short and that provide hyperrectangular decision borders in the feature space, where the borders are parallel to the feature axes. It is not always the best solution to use this kind of rules, as they may not be the best way to describe all systems, but as long as they are as good as other more complex rule sets, they should be preferred as the simplest to understand.

In this chapter we will present a special architecture of a neural network designed for simultaneous feature and instance selection and logical rule extraction. We will present two solutions; one for classification and one for regression tasks. These solutions are based on the decision borders parallel to the feature axes, however if it is required to increase the rule accuracy they allow for an addition of a simple skew (oblique, not parallel to feature axes) part in the rules.

An example of a crisp logical rule parallel to the feature axes:

if $f1 > 20$ and $f2 > 30$ then $classA$

An example of an oblique logical rule:

if $f1 + f2 > 60$ then $classA$

where $f1$ and $f2$ are values of the features. The oblique parts added to the rules can be used as well for classification as regression tasks, but in regression they improve the accuracy more frequently.

18.2 Network Construction and Training

We will call the network a DSRE network (Data Selection and Rule Extraction network). First we present the structure of the network, then logical rule extraction for classification tasks and for regression tasks and finally the instance selection. The first idea of the network was presented in our previous work [199]. In this work we extend its functionalities to process regression tasks, to generate oblique part of the rules and to incorporate instance selection.

An important source of difficulties with feature selection and even more with decompositional logical rule extraction from classical MLP neural network is that the signal propagation through the network is very complex, because particular weights of the hidden neurons are common for each output neuron representing each class. The solution that we consider here is to use dedicated hidden neurons for each class in the classification problem and in the regression problem to split the whole output variable range into several bins and also use separate hidden neurons for each bin, as shown in Fig. 18.1. This will significantly simplify the analysis and will also make the network easier to train, as instead of a multi-class classification problem we will get several single classes (or several two class problems: this class all any other class). Also a specific training algorithm is required and we use the VSS algorithm (see chapter 14), but also Rprop with some adjustments can be used.

The network requires discrete input data. If the data is continuous, it must be discretized prior to the training or at the run-time by an additional network layer.

The basic version of the network uses three layers of neurons. Neurons implement sigmoidal transfer function with the slope β. Initially β=1 and during the network training β gradually increases and finally the transfer functions become step functions. Additionally a penalty term is used to enforce all the weights w to take finally only three possible values (-1,0,1):

$$Error = Error + \sum_{i=0}^{W}(1 - w_i)w_i(1 + w_i) \tag{18.1}$$

where W is the number of weights in the network. The biasses are enforced to take any integer value plus 0.5 (e.g. -2.5, -1.5, -0.5, 0.5, 1.5, etc.) for the hidden layer neurons and only -0.5 or 0.5 for output layer neurons.

At the beginning one hidden neuron is created per class (Neurons N[1,0] and N[1,1] in Fig. 18.1). The second hidden neuron per class (N[1,2] and N[1,3]) is added, if the results with only one neuron are not satisfactory, this is if too many instances are misclassified. Each hidden neuron classifies each cluster of the data, starting from the biggest cluster. What means "too many instances" is to be determined by the user by a parameter that sets the minimal number Mn of instances in a cluster. In a special case $Mn = 1$, what means that the network is required to classify correctly all instances from the training set. However, it is not difficult to oversee that in this case the

network will have poor generalization abilities, so usually some higher number will be preferred.

If the results are still unsatisfactory then the next hidden neuron is added. The number of hidden neurons per a given class should equal the number of the data clusters within this class, which contain at least Mn instances and which cannot be joined together without decreasing the classification accuracy.

18.3 Rule Extraction and Feature Selection for Classification Tasks

First let us consider this network for classification tasks and then we will discuss the regression version, which is based on the classification version with some extensions added.

Logical rules are extracted after the network is trained. Therefore, the rule extraction process does not depend on the algorithm used to train the network.

Each cluster is represented by one disjoined rule generated by the neuron and contains some instances. The order of the clusters expresses the importance of the clusters (how many cases they classify correctly). The clusters can be added with positive weights ($w = 1$) between the hidden and the output neuron and with negative weights ($w = -1$). Positive weight means that the instances in this cluster are included in the class and negative weight means that they are excluded (they are exceptions). For example let us assume that there is a following rule describing the data:

if $f1 > 20$ and not $40 < f1 < 45$ then *class A*

in this case all instances with $f1 > 20$ will go to the first cluster represented by the first hidden neuron with positive weight to the output neuron and all instances with $40 < f1 < 45$, which are the exceptions to this rule, will go to the second cluster represented by the second hidden neuron with negative weight to the output neuron.

Weights as well between input and hidden as between hidden and output neurons that have already been trained are frozen. That is once a cluster is determined it is no longer evaluated and we consider only the remaining data. This makes the training easier and minimizes calculation time. This incremental learning decomposes the task into learning general rules first and then exceptions to these rules instead of trying to modify all rules at once to fit the data.

The network diagram is shown in Fig. 18.1. Each value of a discrete feature is propagated through a separate input neuron. So the number of input neurons equals the sum of the numbers of all distinct values for all feature (either original symbolic values or values that represent particular discretization bins).

If a given value of a certain feature occurs in a given instance, then it is represented by the input signal, which equals 1. For all values, which do not exist in a given

instance, the incoming signals are zero. If a presence of a given value contributes to a given class, the hidden neuron weight will take positive value after the network is trained. If the absence - then the corresponding weight will take negative value. If the attribute is irrelevant to this class then the corresponding weight will be zero.

For symbolic attributes the rules tends to be longer, as each value of the attribute that contributes to a given class must be listed in the rule. For discretized continuous values, we can join together the neighbor values into one and thus for example the original rule generated by the network:

if ($f1 < 20$ or $20 < f1 < 40$ or $40 < f1 < 60$) then *class A*

will become:

if ($f1 < 60$) then *class A*

And as this is in practice a very frequent case, the final rules get much simplified.

The hidden neurons generate M-of-N rules (if M assumptions out of N are satisfied then the condition is true). If the sum of all N inputs of a hidden neuron exceeds its bias, which has the value of M-0.5, then a logical rule is generated. Either the M-of-N rules or the AND/OR rules may describe a given problem more adequately and may be preferred in a given situation. Thus to get the simple AND/OR rules, we must consider the entire attributes and not only their particular values. The relationship between particular values of the same attribute is always OR as it is not possible to have more than one value of one attribute in a single instance. The relationship between the obtained values of different attributes can be AND (then bias equals the number of attributes - 0.5), OR (then bias equals 0.5), or M-of-N (then the bias equals M-0.5)

The output layer performs always OR operations, combining rule conditions into final rules. The obtained rules that are extracted from the data by the analysis of the weights in the trained network are quite straightforward and intuitive.

The features which come mostly with zero weights at least for the most numerous data clusters are natural candidates for removal. In this way feature selection is in a matter of fact performed by logical rule extraction. After the features get removed the network can be left as it is if the removal does not decrease classification accuracy or otherwise it can be retrained without these features.

The way to reject the outliers is by rejecting the clusters with fewer instances than some predefined number. The way to reject instances for the purpose of data condensation is by eliminating the instances on which the network makes the smallest errors, as described in chapter 15.2.

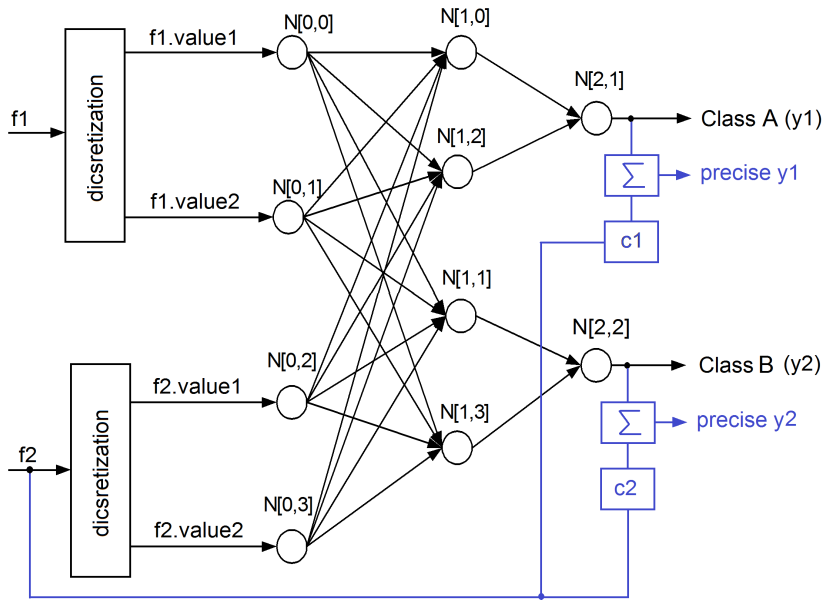


Fig. 18.1. Architecture of the DSRE neural network. The optional oblique rule part is shown in blue.

18.4 Rule Extraction and Feature Selection for Regression Tasks

The presented DSRE network structure can be extended to regression problems. We propose two steps of the extension. The first step - "coarse approximation" uses discretization to convert the regression problem to multiple class classification (the black part of the network in Fig. 18.1). The second step - "fine approximation" additionally uses one or two-variable linear regression inside the discretization bins (the blue part of the network in Fig. 18.1).

The process of DSRE network construction and logical rule extraction for regression problems:

1. Discretize the output data into several bins, temporarily converting the task to the multi-class classification tasks.
2. Discretize the input data in the same way as for classification problem.
3. Train and build the network in the same way as for classification tasks, performing also data compression.
4. Within each discretization bin (each temporary class) consider separately the part of the data represented by each hidden neuron.

a. Find correlation coefficients between each input feature f and the output y
b. Select the input variable with the strongest correlation
c. Approximate the dependence with a linear regression - this will replace the temporary class in the rule. An example illustrated in Fig. 18.1:
if $(0.3 < f1 < 0.5$ and $f2 < -0.5)$ then *class A* (output between 0.4 and 0.5, thus can be assumed 0.45)
will be replaced by:
if $(0.3 < f1 < 0.5$ and $f2 < -0.5)$ then $y1 = 0.45 + c1 \cdot f2$
assuming that the strongest correlation withing this hidden neuron data is between $f1$ and output.
d. Optionally two inputs can be used instead of one if a higher rule accuracy is preferred over simplicity. For example:
if $(0.3 < f1 < 0.5$ and $f2 < -0.5)$ then $y1 = 0.45 + c1 \cdot f2 + c3 \cdot f1$.

5. Perform instance selection to remove outliers as described in the next subsection.

The same oblique rules can be added to the classification version of the network.

18.5 Instance Selection

Two scenarios of instance selection must be distinguished: data compression and noise reduction.

1. Data compression. The first phase of the network training uses sigmoidal transfer functions with the slope β=1. Before going to the second training stage, which increases β, the instances, which cause the smallest network error are found and can be removed. This is done in the same way as in a standard MLP network training and as in the case of standard MLP architectures the method can be used only for classification tasks.
2. Noise reduction. This is done differently than in a standard MLP network. The first hidden neuron, as it was discussed, represents the larger cluster of instances for a given class. The second hidden neuron - the second larger cluster and so on. Finally the last hidden neuron or several last hidden neurons represent very few instances and thus those instances can be considered noise and removed from the dataset. In standard MLP networks instances, on which the trained network makes big errors can be considered noise. In case of DSRE networks, the hidden neurons are added one by one and the instances on which the network makes big errors are first expected to belong to another data cluster and the next hidden neuron is added for them. However, the clusters gradually contain fewer and fewer instances and at a certain point we can decide which instances in the smallest clusters should be rejected as the noisy ones.

18.6 Other Solutions from Literature

18.6.1 Decompositional Rule Extraction from Neural Networks

Decompositional methods extract rules by analyzing the values of weights and propagation of signals in the neural network. Thus the rules directly explain the knowledge representation in the neural network.

Subset Algorithms and M-of-N

Several similar decompositional methods such as SUBSET [200], KT [201], RULE-OUT [202] and Destructive Learning [203] differ only in some details but use the same methodology of rule extraction. The SUBSET method works as follows:

1. Find all combinations p with positive weights to neuron N whose sum exceeds its threshold
2. For each $p = p_1, ..., p_i$
   a. find the set S_n of all combinations of negative weights to N, such that the sum of the weights of p and the weights of $N - n$ exceeds the threshold of N, where n is an element of S_n
   b. for each element $n = n_1, ..., n_j$ create the rule: *if* p_1,...,p_i, *not* n_1,..., *not* n_j *then* N

To overcome the high complexity of SUBSET and to further increase the comprehensibility of a rule system, Towell [200] developed the following M-of-N algorithm:

1. For each neuron, cluster the incoming connections into groups with similar weights
2. Average the weights within each cluster
3. Eliminate the clusters without significant effect on the output of the neuron
4. Re-train the network with frozen weights to optimize biases
5. Form a single rule for each neuron
6. Simplify rules to M-of-N form

NeuroRule and M-of-N3

Neurorule and M-of-N3 are two similar decompositional algorithms developed by Setiono [204]. They share the common network training and rule extraction technique:

1. Train the network until the required accuracy is obtained
2. Remove the redundant connections in the network by pruning while maintaining its accuracy. Steps 1 and 2 can be repeated several times if required
3. Discretize the hidden unit activation values of the pruned network by agglomerative clustering (the neighboring activation values of different input patterns are joined together as long as this does not change the network classification)

4. Extract rules that describe the network outputs in terms of the discretized hidden unit activation values (find any combination of hidden neuron signals that causes the output neuron to fire, i.e. to produce the positive output signal)
5. Generate rules that describe the discretized hidden unit activation values in terms of network inputs (find any combination of inputs that makes the hidden neuron activation within particular discretization interval)
6. Merge the two sets of rules to obtain a set of rules that relates the inputs and outputs of the network

Both the hidden and output neuron use hyperbolic tangent transfer functions. The algorithms require discrete input data. The present value of a given feature is coded as +1 and the absent values as –1. The training process starts with an oversized network that is successively pruned. In the case of M-of-N3, after the small weights are removed, the remaining positive weights are set to +1 and the negative ones to –1. Since the network training starts with random weights, different rule sets can be extracted from the same dataset, depending on the initial weights distribution. When we met Setiono and discussed these issues with him, he admitted that he considers Neurorule the best of the many rule extraction algorithms he created.

Other Decompositional Methods for Classification Tasks

Also a lot of other rule extraction methods from neural networks were proposed. Few examples are provided below. Gupta et al. [205] proposed a GRG method, which extracts rules for data with discrete attributes. Neural networks with one hidden layer are trained and the GRG algorithm is applied to their discretized hidden unit activation values. Setiono et al. [206] proposed a Recursive Rule Extraction algorithm (Re-RX), which generates hierarchical rules from data with discrete and continuous attributes. The RULEX [207] algorithm is based on constrained MLP networks with pairs of sigmoidal functions combined to form ridges or local bumps. Setiono also proposed the FERNN method [208], which extracts oblique rules, but the rules can sometimes be simplified to M-of-N rules. Ozbakir et al. [209] presented a method for rule extraction, which uses differential evolution algorithm for training and ant colony optimization algorithm for extracting logical rules. Kim and Lee [210] proposed an algorithm based on feature extraction and feature combination for neural networks with two hidden layers. Gupta et al. [205] proposed a method, which extracts rules by directly interpreting the strengths of connection weights in a trained network. Krishnan et al., [211] proposed a search technique for rule extraction from neural networks, which by sorting and ordering the input weights to neurons finds the combinations of inputs (and thus the rules) that activate the neuron.

REFANN - Decompositional Rule Extraction for Regression Tasks

Setiono et. al. [212] proposed a method called REFANN for rule extraction from neural networks trained for regression problems. REFANN first prunes the network to

limit its size and then extracts linear rules by approximating the hidden unit activation functions by piecewise linear functions as shown in Fig. 18.2.

REFANN attempts to provide an explanation for the network outputs by replacing the nonlinear mapping of a pruned network by a set of linear regression equations. Using the weights of a trained network, the input space is divided into a small number of subregions such that the prediction for the samples in the same subregion can be computed by a single linear equation. REFANN approximates the nonlinear hyperbolic tangent activation function of the hidden units using a simple three-piece or five-piece linear function. It then generates rules in the form of linear equations from the trained network. The conditions in these rules divide the input space into one or more subregions. For each subregion, a linear equation that approximates the network output is generated.

In general, a rule condition is defined in terms of the weighted sum of the inputs, which corresponds to an oblique hyperplane in the input space. This type of rule condition can be difficult for the users to interpret. In some cases, the oblique hyperplanes can be replaced by hyperplanes that are parallel to the axes by employing a classification method such as C4.5 in the optional step.

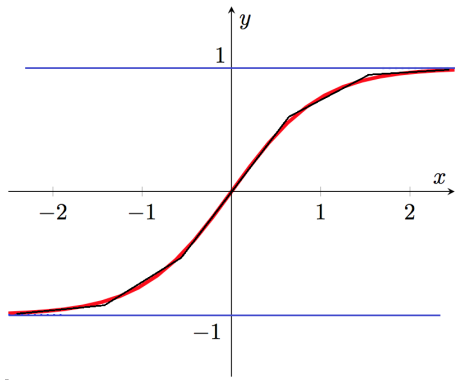


Fig. 18.2. The idea of linear-piece approximation of hyperbolic activation function in the REFANN method.

18.6.2 Pedagogical Rule Extraction from Neural Networks

Pedagogical rules are obtained by mapping the input-output relationships as closely as possible to the prediction of neural network. It is usually easier to obtain simpler rules in this way, however the neural network is treated as a black box, so we do not get an explanation of its inner knowledge representation.

Pedagogical rule extraction only build the logical rules using the neural network prediction, but they do not explain how the neural network reached the conclusions. Thus these rules may provide less solid proof for users that the decision the network took is correct. For that reason we only very shortly review some of them just to give the reader a general outlook on how the methods work.

Validity Interval Analysis (VIA)

An example of global methods is Validity Interval Analysis (VIA) proposed by Thrun [213]. The key idea in VIA is to attach intervals to the activation range of each neuron (or a subset of all neurons), such that the network activation must lie within these intervals, called validity intervals I. VIA checks whether there exists a set of network activations inside the validity intervals. It does this by iteratively refining the validity intervals, excluding activations that are probably inconsistent with other intervals. The obtained rules are prepositional if-then rules, where the precondition is given by a set of intervals for the individual input values and the output is a single target category. Rules of this type can be written as:

if *input is contained in the hypercube I* then *class is C* (or shortly: I then C)

Two types of approaches can be distinguished: specific-to-general and general-to-specific. In a specific-to-general approach we start with specific rules that are easy to verify and gradually generalize them by enlarging the corresponding validity intervals. In a general-to-specific approach we start from rules like "everything is in class C" and then a new rule can be generated by splitting the hypercube spanned by the old rule.

GEX and GenPar

Two similar pedagogical rule extraction methods called GEX and Genoa were proposed in [214] and [215]. Thirst the MLP network predicts the class of all the instances. Then a genetic algorithm-based rule extraction module generates rules for the class, which was predicted by the neural network. The advantage of that approach is that the neural network by its nature removes some noise from the data, thus the rules get simpler, better reflect the real properties of the data and are more comprehensive. The process is performed as follows: first a rule set is encoded in the chromosome. Then the training instances are applied to the rule set and the neural network. Each individual in the genetic population is evaluated with respect to its accuracy (number of misclassified examples) and comprehensibility (number of rules and premises). Then the genetic algorithm searches for the best individuals by calculating the total fitness value for each of them and performs the typical genetic operations of crossover and mutation to produce the next generation. This method follows the key idea of the TREPAN algorithm, however instead of using decision trees, it uses genetic algorithms to generate and optimize the logical rules.

18.6.3 Hybrid Rule Extraction from Neural Networks

The idea of hybrid methods is that they use partially the compositional and partially the pedagogical approach. An example of a hybrid method is FERNN (Fast Extraction of Rules from Neural Networks) proposed by Setiono [208]. FERNN extracts the rules without the time consuming weight pruning. First the identification of useful hidden units is performed based on the information contained in these units. For this purpose C4.5 is employed (pedagogical part). However, identification of relevant connections between input and hidden units is based on magnitudes of the weights (decompositional part). In this way FERNN is really a mixed algorithm.

18.7 Conclusions

Several methods of rule extraction from trained neural networks were proposed in the literature. The decompositional approach is most interesting, as it allows directly to understand (at least to some degree) how the network works and how the decisions were taken. However, the problem with most approaches is that the rules are only some approximation of the real connections in the neural network and therefore they describe the network only with a limited accuracy. Thus we proposed a method called DSRE, which allows for exact description of the network without excessive complexity of the generated rules.

The DSRE network combines the advantages of MLP neural networks with the possibility of extracting simple rules in a comprehensive way. The training algorithm is based on typical neural network algorithms with additional penalty term to enforce the integer weight and with variable sigmoid slope to finally lead the network to a state, where extracting logical rules is straightforward. The hidden neurons generate the M-of-N rules, but they can frequently be reduced to simpler to understand AND + OR operations.

The quality of results on the popular benchmark datasets is comparable with the best results obtained from other methods, while there is an additional functionality of obtaining logical rules, which perfectly match the network properties. The process of logical rule extraction automatically performs feature selection and instance selection as outlier detection and removal. Instance selection as data condensation can be performed in the same way as in standard MLP networks by eliminating these instances, on which the network makes the smallest errors in classification tasks before increasing the sigmoid slope. Additionally an oblique part can be added to the rules what increases the prediction quality especially in regression problems.

Chapter 19
Summary and Conclusions

A lot of instance selection methods have been developed, especially in recent years. In this book we have presented three main groups of methods and some representative solutions in each of them.

Instance selection approaches can be classified using the following criteria:

1. Purpose of the selection:
   - condensation methods
   - noise filters
2. Target:
   - single-output classification
   - single-output regression
   - multi-output classification
   - multi-output regression
   - multi-output mixed classification and regression
3. Groups of the algorithms:
   - similarity based
   - evolutionary
     - single-objective
     - multi-objective
   - embedded into learning of predictive models
   - hybrid and others
4. Joint processes:
   - only instance selection
   - instance selection together with feature selection
   - instance selection together with feature selection and logical rule extraction

In many of the above items we have also presented our own solutions or improvements to the existing ones. Obviously, each of the groups has advantages and disadvantages and choosing a given method and its parameters should depend on two factors: the data properties and the user preferences. The preferences can include the following parameters:

1. compression vs accuracy trade-off
2. algorithm running time
3. algorithm complexity, easiness to incorporate it into our software
4. understandability of the accept/reject decision made by the algorithm

To maximize the first criterion, we can to try an ensemble of the multi-objective evolutionary-based instance selection methods with multiply Pareto front merging, data partitioning and with an inner evaluation model being the same as the final predictor. However, this method will be very complex to implement, will have high running time (other inner evaluator than k-NN and ensemble). Moreover, the ensembles of the MEISR2 methods have not been tested by us yet, so it is not guarantied that this will perform better than a single algorithm, but it is surely worth trying. The next point is that evolutionary methods do not provide an explanation, why particular decision was made. This is frequently not concern, but for some users it cam be important.

To minimize the algorithm running time, we can use CNN with some method of accelerating distance matrix calculations. In regression tasks we can also easily partition the data for T-CNN. But again CNN is usually not the best method and the simplicity of this solution will not be excellent, because of the operations performed with the distance matrix. The understandability will be better than of the multi-objective evolutionary-based solution, but still not perfect due to the random order in which CNN processes data.

To allow easy implementation the algorithm has to be simple. Obviously a simple method will not perform very well and will not be the fastest.

The DROP3 and DROP5 algorithms are very good in terms of understandability, as they perform instance selection based on defined assumptions about the data properties. They usually perform better than ENN followed by CNN and also than some other similarity-based methods, but in most cases not so well as the evolutionary instance selection and their running time is relatively high comparing to other methods.

In practical implementations, we usually need to find a reasonable compromise between all the criteria. An example of the compromise, which we proposed is to use k-NN with optimal k and an appropriate weighting scheme as an inner evaluator of the evolutionary instance selection.

In Fig. 19.1 we show the general trend in different approaches to instance selection in regression tasks. The lowest grey (or grey-green) line represents an untested solution, so it is shown with dotted line, and the position of the line is only based on our assumptions, in practice so good performance it is not guaranteed. Moreover, for specific datasets frequently some earlier method on the list obtained better results than the later methods. Another advantage of the earlier methods is their simplicity

and frequently also lower computational cost. For example introducing the same inner evaluator as the final predictor (if it is not k-NN) to evolutionary-based instance selection increases the computational cost by about three orders of magnitude and adding additionally ensembles by another order of magnitude. For that reason going below the orange line in Fig. 19.1 may be frequently impractical.

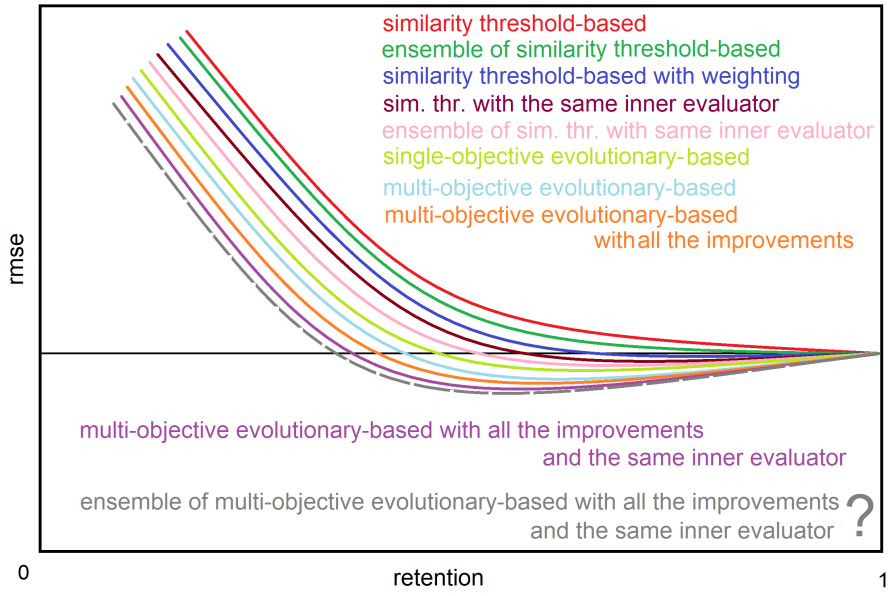


Fig. 19.1. Trends of average Pareto fronts obtained with different instance selection approaches for regression tasks.

There are some differences in the instance selection performance for classification tasks:

- Much stronger compression can be obtained in classification than in regression tasks.
- Using instance selection, the prediction accuracy can be less frequently improved in classification tasks than in regression tasks.
- Ensemble methods allow to adjust the accuracy-compression balance in classification but less frequently allow to improve both criteria at the same time than in regression tasks.

Besides the listed differences, the general trends for instance selection in classification tasks are similar to those in regression tasks shown in Fig. 19.1 (assuming that there is $1 - accuracy$ on the vertical axis).

Other topics discussed in the book included joint feature and instance selection and instance selection embedded into neural network learning process.

Both feature selection and instance selection allow to reduce the data size and optimally also to improve prediction results. As in instance selection always noise should be removed first and data condensation performed as the next step, in joint feature and instance selection also noise should be removed first. For that purpose we must identify the main source of noise and in most typical data it is rather associated with features than with instances. For that reason in these cases feature selection should be performed first. If the noise is equally distributed in features and instances then iterative approaches work well.

Neural networks by their nature perform feature weighting and differently respond to different instances. However, to make a practical use of this, some modifications to the training procedure must be made, as special dynamic error functions, controlling the speed of learning and removing some instances at appropriate time or designing a special neural network structures especially dedicated to data selection and logical rule extraction, as was presented in the last chapter. The accuracy of data selection performed by neural networks may be comparable to that obtained with similarity-based methods, but there are two other advantages: we do not have to add a separate instance selection and/or feature selection step to the process and by analyzing which instances/features were rejected by what mechanism during network learning we can better understand the dataset and how the network processes it. Disadvantages of this solution may comprise sometimes higher computational cost and difficult implementation.

We presented important problems of data selection and many solutions to them, including those developed or co-developed by the author, which were discussed in more detail. We believe that this book was a useful source of information on data reduction and noise removal with instance selection and that it allowed the reader to get a good insight into this field.

References

1. D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, 1989.
2. S. García, J. Luengo, and F. Herrera. *Data preprocessing in data mining*. Springer, 2015.
3. S. García, J. Derrac, J. Cano, and F. Herrera. Prototype selection for nearest neighbor classification: Taxonomy and empirical study. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(3):417–435, 2012.
4. F. R. Hampel, E. M. Ronchetti, P. J. Rousseeuw, and W. A. Stahel. Robust statistics: the approach based on influence functions. *Wiley Series in Probability and Statistics*, 2005.
5. M. Blachnik. Instance selection for classifier performance estimation in meta learning. *Entropy*, 19(11):583, 2017.
6. R. Barandela, F. J. Ferri, and J. S. Sánchez. Decision boundary preserving prototype selection for nearest neighbor classification. *International Journal of Pattern Recognition and Artificial Intelligence*, 19(06):787–806, 2005.
7. A. Rusiecki, M. Kordos, T. Kamiński, and K. Greń. Training neural networks on noisy data. In *International Conference on Artificial Intelligence and Soft Computing*, pages 131–142. Springer, 2014.
8. J. Alcalá-Fdez, A. Fernández, J. Luengo, J. Derrac, and S. García. KEEL data-mining software tool: Data set repository, integration of algorithms and experimental analysis framework. *Multiple-Valued Logic and Soft Computing*, 17(2-3):255–287, 2011.
9. S. García, J. Derrac, J. R. Cano, and F. Herrera. Prototype selection for nearest neighbor classification: Taxonomy and empirical study. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(3):417–435, March 2012.
10. L. I. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley, 2004.
11. P. E. Hart. The condensed nearest neighbor rule (corresp.). *Information Theory, IEEE Transactions on*, 14(3):515–516, may 1968.
12. D. R. Wilson and T. R. Martinez. Reduction techniques for instance-based learning algorithms. *Machine Learning*, 38(3):257–286, 2000.
13. J. S. Sánchez, F. Pla, and F. J. Ferri. Prototype selection for the nearest neighbour rule through proximity graphs. *Pattern Recognition Letters*, 18(6):507 – 513, 1997.
14. I. Tomek. An experiment with the edited nearest-neighbor rule. In *IEEE*, volume 6 of *Transactions on Systems, Man, and Cybernetics*, page 448–452, 1976.
15. D.W. Aha, D. Kibler, and M.K. Albert. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.

16. R. M. Cameron-Jones. Instance selection by encoding length heuristic with random mutation hill climbing. In *The Eighth Australian Joint Conference on Artificial Intelligence*, pages 99–106, 1995.
17. D. B. Skalak. Prototype and feature selection by sampling and random mutation hill climbing algorithms. *Machine Learning Proceedings*, 10–13:293–301, 1994.
18. J. Leskovec, A. Rajaraman, and J. D. Ullman. Mining of massive datasets. *Cambridge University Press*, 2nd edition, 2014.
19. M. Grochowski and N. Jankowski. Comparison of instance selection algorithms ii. results and comments. In *International Conference on Artificial Intelligence and Soft Computing*, pages 580–585. Springer, 2004.
20. T. Kohonen. Learning vector quantization for pattern recognition. *Technical Report TKK-F-A601*, 1986.
21. H. Brighton and Ch. Mellish. Advances in instance selection for instance-based learning algorithms. *Data Mining and Knowledge Discovery*, 6(2):153–172, 2002.
22. E. Leyva, A. González, and R. Pérez. Three new instance selection methods based on local sets: A comparative study with several approaches from a bi-objective perspective. *Pattern Recognition*, 48(4):1523 – 1537, 2015.
23. E. Marchiori. Class conditional nearest neighbor for large margin instance selection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(2):364–370, Feb 2010.
24. G. D. C. Cavalcanti, T. I. Ren, and C. L. Pereira. Adaptive threshold-based instance selection algorithm. *Expert Systems with Applications*, 40:6894—-6900, 2013.
25. L. Yang and Q. Zhu. Constraint nearest neighbor for instance reduction. *Soft Computing*, First Online: 22 March 2019, 2019.
26. I. Ben-Gal. Outlier detection. In *Data Mining and Knowledge Discovery Handbook*, pages 131–146. Springer, 2005.
27. M. Amer and M. Goldstein. Nearest-neighbor and clustering based anomaly detection algorithms for rapidminer. In *RCOMM 2012*, 2012.
28. I. Triguero, D. Peralta, J. Bacardit, S. García, and F. Herrera. Mrpr: A mapreduce solution for prototype reduction in big data classification. *Neurocomputing*, 150(Part A):331–345, 2015.
29. N. García-Pedrajas and A. Haro-García. Boosting instance selection algorithms. *Knowledge-Based Systems*, 67:342–360, 2014.
30. Á. Arnaiz-González, J. F. Díez-Pastor, J. J. Rodríguez, and C. García-Osorio. Instance selection for regression: Adapting drop. *Neurocomputing*, 201:66–81, 2016.
31. M. Blachnik and M. Kordos. Instance selection in rapidminer. *RapidMiner : data mining use cases and business analytics applications*, 2014.
32. M. Blachnik and M. Kordos. Information selection and data compression rapidminer library. *Machine Intelligence and Big Data in Industry*, 2016.
33. L. Nanni and A. Lumini. Prototype reduction techniques: A comparison among different approaches. *Expert Systems with Applications*, 38(9):11820–11828, 2011.
34. M. Kordos and M. Blachnik. Instance selection with neural networks for regression problems. In Alessandro E.P. Villa, Włodzisław Duch, Péter Érdi, Francesco Masulli, and Günther Palm, editors, *Artificial Neural Networks and Machine Learning - ICANN 2012*, volume 7553 of *Lecture Notes in Computer Science*, pages 263–270. Springer Berlin Heidelberg, 2012.
35. M. Kordos, S. Białka, and M. Blachnik. Instance selection in logical rule extraction for regression problems. In *Lecture notes in computer science*, volume 7895, pages 167–175. Springer Berlin Heidelberg, 2013.
36. I. Rodriguez-Fdez, M. Mucientes, and A. Bugarin. An instance selection algorithm for regression and its application in variance reduction. In *Fuzzy Systems (FUZZ), 2013 IEEE International Conference on*, pages 1–8, July 2013.
37. D. L. Wilson. Asymptotic properties of nearest neighbor rules using edited data. *Systems, Man and Cybernetics, IEEE Transactions on*, SMC-2(3):408–421, July 1972.

38. Á. Arnaiz-González, M. Blachnik, M. Kordos, and C. García-Osorio. Fusion of instance selection methods in regression tasks. *Information Fusion*, 30:69–79, 2016.
39. Á. Arnaiz-González, J. F. Díez-Pastor, J. J. Rodríguez, and C. I. García-Osorio. Instance selection for regression by discretization. *Expert Systems with Applications*, 54:340–350, 2016.
40. J. Zhangg, Y. S. Yim, and J. Yang. Intelligent selection of instances for prediction functions in lazy learning algorithms. In David W. Aha, editor, *Lazy Learning*, pages 175–191. Springer Netherlands, 1997.
41. A. Guillen, L. J. Herrera, G. Rubio, H. Pomares, A. Lendasse, and I. Rojas. New method for instance or prototype selection using mutual information in time series prediction. *Neurocomputing*, 73(10-12):2030 – 2038, 2010. Subspace Learning / Selected papers from the European Symposium on Time Series Prediction.
42. M. Božić, M. Stojanović, Z. Stajić, and N. Floranović. Mutual information-based inputs selection for electric load time series forecasting. *Entropy*, 15(3):926–942, 2013.
43. M. B. Stojanović, M. M. Božić, M. M. Stanković, and Z. P. Stajić. A methodology for training set instance selection using mutual information in time series prediction. *Neurocomputing*, 141:236 – 245, 2014.
44. S. H. Son and J. Y. Kim. Data reduction for instance-based learning using entropy-based partitioning. In *International Conference on Computational Science and Its Applications*, pages 590–599. Springer, 2006.
45. A. Arsen, H. Waseem, and J. Seokhee. Stimuli-magnitude-adaptive sample selection for data-driven haptic modeling. *Entropy*, 18(6):222, 2016.
46. M. Kordos, A. Rusiecki, and M. Blachnik. Noise reduction in regression tasks with distance, instance, attribute and density weighting. In *Cybernetics (CYBCONF), 2015 IEEE 2nd International Conference on*, pages 73–78. IEEE, 2015.
47. D. Wettschereck and et. al. A review and empirical evaluation of feature weighting methods for a class of lazy learning algorithms. *Artificial Intelligence Review*, 11(1-5):273–314, 1997.
48. J. Gou and et. al. A new distance-weighted k-nearest neighbor classifier. *Journal of Information & Computational Science*, 9(6):1429–1440, 2012.
49. R. Polikar. Ensemble based systems in decision making. *Circuits and Systems Magazine, IEEE*, 6(3):21–45, 2006.
50. J. Maudes, J. J. Rodríguez, C. García-Osorio, and N. García-Pedrajas. Random feature weights for decision tree ensemble construction. *Information Fusion*, 13(1):20 – 30, 2012.
51. G. Brown, J. Wyatt, R. Harris, and X. Yao. Diversity creation methods: a survey and categorisation. *Information Fusion*, 6(1):5 – 20, 2005. Diversity in Multiple Classifier Systems.
52. T. Woloszynski, M. Kurzynski, P. Podsiadlo, and G. W. Stachowiak. A measure of competence based on random classification for dynamic ensemble selection. *Information Fusion*, 13(3):207 – 213, 2012.
53. Z.-H. Zhou. Ensemble methods: Foundations and algorithms. *CRC Press*, 2012.
54. M. Woźniak, M. Graña, and E. Corchado. A survey of multiple classifier systems as hybrid systems. *Information Fusion*, 16:3 – 17, 2014. Special Issue on Information Fusion in Hybrid Intelligent Fusion Systems.
55. Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. *Morgan Kaufmann Publishers Inc.*, page 148–156, 1996.
56. Y. Freund and R. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55(119), 1997.
57. T. K. Ho. Random decision forests (pdf). In *Proceedings of the 3rd International Conference on Document Analysis and Recognition*, QC, 14–16 August 1995, page 278–282, 2016.
58. D. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992.
59. R. Scherer. Designing boosting ensemble of relational fuzzy systems. *International Journal of Neural Systems*, 20:381–388, 2010.

60. M. Korytkowski, R. Nowicki, L. Rutkowski, and R. Scherer. Adaboost ensemble of dcog rough–neuro–fuzzy systems. In *International Conference on Computational Collective Intelligence*, pages 62–71, 2011.
61. Ch. Ju, A. Bibaut, and M. J. van der Laan. The relative performance of ensemble methods with deep convolutional neural networks for image classification. *arXiv:1704.01664v1*, 2017.
62. J. D. Wichard and M. Ogorzalek. Time series prediction with ensemble models. In *2004 IEEE International Joint Conference on Neural Networks*, IEEE Cat. No.04CH37541, 2004.
63. L. Breiman. Bagging predictors. *Mach. Learn.*, 24(2):123–140, August 1996.
64. M. Blachnik. Ensembles of instance selection methods. a comparative study. *International Journal of Applied Mathematics and Compututer Science*, 21(1):151–168, 2019.
65. C. I. García-Osorio, A. de Haro-García, and N. García-Pedrajas. Democratic instance selection: A linear complexity instance selection algorithm based on classifier ensemble concepts. *Artificial Intelligence*, 174(5-6):410–441, 2010.
66. M. Blachnik and M. Kordos. Bagging of instance selection algorithms. In Leszek Rutkowski, Marcin Korytkowski, Rafał Scherer, Ryszard Tadeusiewicz, Lotfi A. Zadeh, and Jacek M. Zurada, editors, *Artificial Intelligence and Soft Computing*, volume 8468 of *Lecture Notes in Computer Science*, pages 40–51. Springer International Publishing, 2014.
67. H. Liu. *Computational Methods of Feature Selection*. Chapman and Hall, 2007.
68. U. Stanczyk and L. C. Jain. *Feature Selection for Data and Pattern Recognition*. Springer, 2015.
69. C. Uribe and C. Isaza. Expert knowledge-guided feature selection for data-based industrial process monitoring. *Rev. Fac. Ing. Univ. Antioquia*, 65:112–125, 2012.
70. M. Kordos, M. Blachnik, J. Kozłowski, M. Perzyk, O. Bystrzycki, M. Gródek, A. Byrdziak, and Z. Motyka. A hybrid system with regression trees in steelmaking process. In *Lecture Notes in Artificial Intelligence - HAIS*, page 222–229, 2011.
71. W. Duch, T. Wieczorek, J. Biesiada, and M. Blachnik. Comparison of feature ranking methods based on information entropy. In *IEEE*, volume 2 of *International Joint Conference on Neural Networks*, 2004.
72. L. Zhang, C. Chen, J. Bu, and X. He. A unified feature and instance selection framework using optimum experimental design. In *IEEE Trancaction on Image Processing*, volume 21, may 2012.
73. D. Fragoudis, D. Meretakis, and S. Likothanassis. Integrating feature and instance selection for text classification. In *Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 501–506. ACM, New York, 2002.
74. J. T. de Souza, R. A. F. do Carmo, and G. A. C. de Lima. On the combination of feature and instance selection. *Machine Learning*, page 438, 2010.
75. C. F. Tsai, W. Eberle, and C. Y. Chu. Genetic algorithms in feature and instance selection. *Knowledge-Based Systems*, 39:240–247, 2013.
76. J. Derrac. Enhancing evolutionary instance selection algorithms by means of fuzzy rough set based feature selection. *Information Sciences*, 186:73–92, 2012.
77. Y. Tan, C. Yu, S. Zheng, and K. Ding. Introduction to fireworks algorithm. *International Journal of Swarm Intelligence Research*, 4(4):39–71, 2013.
78. M. E. H. Sadati and J. B. Mohasefi. The application of imperialist competitive algorithm for fuzzy random portfolio selection problem. *International Journal of Computer Applications*, 79:10–14, 2013.
79. K. James. Particle swarm optimization. *Encyclopedia of machine learning*, pages 760–766, 2011.
80. Y. Xin-She. Bat algorithm for multi-objective optimisation. *International Journal of Bio-Inspired Computation*, 3(5):267–274, 2011.
81. Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1992.
82. C. K. Ting. On the mean convergence time of multi-parent genetic algorithms without selection. *Advances in Artificial Life*, pages 403–412, 2005.

83. Z. C. Zavoianu and et. al. Performance comparison of generational and steady-state asynchronous multi-objective evolutionary algorithms for computationally-intensive problems. *Knowledge-Based Systems*, 87:47—60, 2015.
84. J. R. Cano, F. Herrera, and M. Lozano. Using evolutionary algorithms as instance selection for data reduction in kdd: An experimental study. *IEEE Transactions on Evolutionary Computations*, 7(6), 2003.
85. J. R. Cano, F. Herrera, and M. Lozano. Instance selection using evolutionary algorithms: An experimental study. *Advanced Information and Knowledge Processing*, pages 127–152, 2004.
86. E. J. Eshelman. *The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination, The First Workshop on Foundations of Genetic Algorithms*. Morgan Kaufmann, 1991.
87. M. A. Potter and K. A. D. Jong. A cooperative coevolutionary approach to function optimization. In *Proceedings of the International Conference on Evolutionary Computation*, The Third Conference on Parallel Problem Solving from Nature, page 249–257, 1994.
88. X. Peng and Y. Wu. Enhancing cooperative coevolution with selective multiple populations for large-scale. *Global Optimization Complexity*, 2018.
89. A. Konak, D. Coit, and A. Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering and System Safety*, 91:992—1007, 2006.
90. F. Rudziński. An application of generalized strength pareto evolutionary algorithm for finding a set of non-dominated solutions with high-spread and well-balanced distribution in the logistics facility location problem. In *International Conference on Artificial Intelligence and Soft Computing*, pages 439–450. Springer, 2017.
91. R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.
92. W. K. Mashwant. Enhanced versions of differential evolution: state-of-the-art survey. *International Journal of Computing Science and Mathematics*, 5(2):107–126, 2014.
93. J. D. Schaffer. Multiple objective optimization with vector evaluated genetic algorithm. In *Proceeding of the First International Conference of Genetic Algorithms and Their Application*, pages 93–100, 1985.
94. K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.
95. Y. Yuan, H. Xu, and B. Wang. An improved nsga-iii procedure for evolutionary many-objective optimization. In *2014 Annual Conference on Genetic and Evolutionary Computation*, ACM, pages 661–668, 2014.
96. H. Liu and H. Motoda. Instance selection and construction for data mining. *Springer Science & Business Media*, 2013.
97. M. Kordos and A. Rusiecki. Improving mlp neural network performance by noise reduction. In *Lecture notes in computer science*, volume 8273, pages 133–144, 2013.
98. J. Tolvi. Genetic algorithms for outlier detection and variable selection in linear regression models. *Soft Computing*, 8(8):527–533, 2004.
99. N. García-Pedrajas, J. A. R. Del Castillo, and D. Ortiz-Boyer. A cooperative coevolutionary algorithm for instance selection for instance-based learning. *Machine Learning*, 78(3):381–420, 2010.
100. M. Antonelli, P. Ducange, and F. Marcelloni. Genetic training instance selection in multiobjective evolutionary fuzzy systems: A coevolutionary approach. *Fuzzy Systems, IEEE Transactions on*, 20(2):276–290, April 2012.
101. J. Derrac, I. Triguero, S. García, and F. Herrera. Integrating instance selection, instance weighting, and feature weighting for nearest neighbor classifiers by coevolutionary algorithms. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(5):1383–1397, 2012.
102. N. García-Pedrajas and J. Pérez-Rodríguez. Multi-selection of instances: A straightforward way to improve evolutionary instance selection. *Applied Soft Computing*, 12:3590–3602, 2012.

103. A. de Haro-García, J. Perez-Rodríguez, and N. García-Pedrajas. Combining three strategies for evolutionary instance selection for instance-based learning. *Swarm and Evolutionary Computation*, 42:160—172, 2018.
104. M. Kordos, M. Wydrzyński, and K. Łapa. Obtaining pareto front in instance selection with ensembles and populations. In *ICAISC, June 2018*, volume 10841 of *Lecture Notes in Artificial Intelligence*, pages 438–448, 2018.
105. M. Kordos and K. Łapa. Multi-objective evolutionary instance selection for regression tasks. *Entropy*, 20(10):746, 2018.
106. M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.
107. M. Santha and U. V. Vazirani. Generating quasi-random sequences from semi-random sources. *Journal of Computer and System Sciences*, 33(1):75–87, 1986.
108. W. Pan, K. Li, M. Wang, J. Wang, and B. Jiang. Adaptive randomness: a new population initialization method. *Mathematical Problems in Engineering*, 2014, 2014.
109. S. Rahnamayan and G. G. Wang. Center-based sampling for population-based algorithms. In *Evolutionary Computation, 2009. CEC'09. IEEE Congress on*, pages 933–938. IEEE, 2009.
110. K. Javed, R. Gouriveau, and N. Zerhouni. Sw-elm: A summation wavelet extreme learning machine algorithm with a priori parameter initialization. *Neurocomputing*, 123:299–307, 2014.
111. Y. Saka. Latinized, improved lhs, and cvt point sets in hypercubes. *International Journal of Numerical Analysis and Modeling*, 4(3-4): 729–743, 2007.
112. L. Kallel and M. Schoenauer. Alternative random initialization in genetic algorithms. In *ICGA*, pages 268–275, 1997.
113. K. Łapa, K. Cpałka, and Y. Hayashi. Hybrid initialization in the process of evolutionary learning. In *International Conference on Artificial Intelligence and Soft Computing*, pages 380–393. Springer, 2017.
114. M. Kordos and A. Rusiecki. Reducing noise impact on mlp training. *Soft Computing*, 20(1):49–65, 2016.
115. A. Rosales-Pérez, S. García, J. A. Gonzalez, C. A. Coello, and F. Herrera. An evolutionary multiobjective model and instance selection for support vector machines with pareto-based ensembles. *IEEE Transactions on Evolutionary Computation*, 21(6):863–877, 2017.
116. H. J. Escalante, M. Marin-Castro, A. Morales-Reyes, M. Graff, A. Rosales-Perez, M. Montes y Gomez, C. A. Reyes, and J. A. Gonzalez. Mopg: a multi-objective evolutionary algorithm for prototype generation. *Pattern Analysis and Applications*, 20(1):33–47, 2017.
117. C. C. Aggarwal and C. C. Reddy. Data clustering: Algorithms and applications. *Chapman and Hall/CRC*, 2013.
118. M. Kordos, Ł. Czepielik, and M. Blachnik. Data set partitioning in evolutionary instance selection. *Lecture Notes in Computer Science*, 11314:631–641, 2018.
119. A. Fouand Aui and A. Nashwa Nageh. Differential evolution algorithm with space partitioning for large scale optimization problems. *I.J. Intelligent Systems and Applications*, 11:49–59, 2015.
120. G. Dun-wei and Z. Yong. Multi-population genetic algorithms with space partition for multi-objective optimization problems. *IJCSNS International Journal of Computer Science and Network Security*, 6(2a), 2006.
121. I. Czarnowski. Cluster-based instance selection for machine classification. *Knowledge and Information Systems*, 30(1):113–133, 2012.
122. I. Czarnowski and P. Jędrzejowicz. Cluster-based instance selection for the imbalanced data classification. In *International Conference on Computational Collective Intelligence*, pages 191–200. Springer, 2018.
123. J. R. Cano, F. Herrera, and M. Lozano. A study on the combination of evolutionary algorithms and stratified strategies for training set selection in data mining. *Technical Report SCI2S-2004-05*, 2004.

124. M. Kordos and A. Kłos-Witowska. Increasing speed of genetic algorithm-based instance selection. In *The 9th IEEE International Conference on Intelligent Data Acquisition and Advanced Computing Systems*, volume 2, pages 90–595, 2017.
125. G. Hamerly and J. Drake. Accelerating lloyd's algorithm for k-means clustering. *Partitional clustering algorithms*, page 41–78, 2015.
126. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509, 1975.
127. A. Gionis, I. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th Very Large Database (VLDB) Conference*, 1999.
128. J. A. Pérez-Benítez, J. L. Pérez-Benítez, and J. H. Espina-Hernández. Novel data condensing method using a prototype's front propagation algorithm. *Engineering Applications of Artificial Intelligence*, 39:181–197, 2015.
129. J. Pérez-Rodríguez, A. G. Arroyo-Peña, and N. García-Pedrajas. Simultaneous instance and feature selection and weighting using evolutionary computation: Proposal and study. *Applied Soft Computing*, 37:416–443, 2015.
130. L. I. Kuncheva and L. C. Jain. Nearest neighbor classifier: simultaneous editing and feature selection. *Pattern Recognition Letters*, 20(11):1149–1156, 1999.
131. J. H. Chen, H. M. Chen, and S. Y. Ho. Design of nearest neighbor classifiers: multiobjective approach. *International Journal of Approximate Reasoning*, 40(1):3–22, 2005.
132. F. Ros, S. Guillaume, M. Pintore, and J. R. Chrétien. Hybrid genetic algorithm for dual selection. *Pattern Analysis & Applications*, 11(2):179–198, 2008.
133. H. Ishibuchi and T. Nakashima. Multi-objective pattern and feature selection by a genetic algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1069–1076, 2000.
134. J. F. Ramirez-Cruz, O. Fuentes, V. Alarcon-Aquino, and L. Garcia-Banuelos. Instance selection and feature weighting using evolutionary algorithms. In *15th International Conference on Computing (CIC '06)*, pages 73–79, 2006.
135. J. Derrac, S. García, and F. Herrera. Instance and feature selection based on cooperative coevolution with nearest neighbor rule. *Pattern Recognition*, 43(2010):2082–2105, 2009.
136. J. Derrac, S. García, and F. Herrera. A first study on the use of coevolutionary algorithms for instance and feature selection. In *HAIS*, volume 5572 of *LNAI*, page 557–564, 2010.
137. B. Sierra, E. Lazkano, I. Inza, M. Merino, P. Larrañaga, and J. Quiroga. Prototype selection and feature subset selection by estimation of distribution algorithms. *Lecture Notes in Computer Science*, 2101/2001:20–29, 2001.
138. J. T. Souza, R. A. F. Carmo, and G. A. L. Campos. A novel approach for integrating feature and instance selection. In *Proceedings of the International Conference on Machine Learning and Cybernetics*, pages 374–379, 2008.
139. N. García-Pedrajas, J. A. R. del Castillo, and D. Ortiz-Boyer. A cooperative coevolutionary algorithm for instance selection for instance-based learning. *Mach Learn*, 78:381–420, 2010.
140. N. García-Pedrajas, A. de Haro-García, and J. Pérez-Rodríguez. A scalable memetic algorithm for simultaneous instance and feature selection. *Evolutionary Computation*, 22(1):1–45, 2014.
141. A. Hyunchul, K. Kyoung-jae, and H. Ingoo. Simultaneous optimization of feature weighting and instance selection in case-based reasoning systems using genetic algorithms. In *9th Asia-Pacific DSI Conference*, 2004.
142. M. Kordos, Á. Arnaiz-González, and C. García-Osorio. Instance selection for multi-output regression problems. *submitted to Neurocomputing in July 2018*, 2019.
143. E. Spyromitros-Xioufis, G. Tsoumakas, W. Groves, and I. Vlahavas. Multi-target regression via input space expansion: treating targets as inputs. *Machine Learning*, 104(1):55–98, 2016.
144. T. Aho, B. Ženko, S. Džeroski, and T. Elomaa. Multi-target regression with rule ensembles. *Journal of Machine Learning Research*, 13:2367–2407, 2012.
145. H. Borchani, G. Varando, C. Bielza, and P. Larrañaga. A survey on multi-output regression, wiley interdisciplinary reviews. *Data Mining and Knowledge Discovery*, 5(5):216–233, 2015.

146. Z. Han, Y. Liu, J. Zhao, and W. Wang. Real time prediction for converter gas tank levels based on multi-output least square support vector regressor. *Control Engineering Practice*, 20(12):1400 – 1409, 2012.
147. D. Kocev, S. Džeroski, M.-D. White, G.-R. Newell, and P. Griffioen. Using single- and multi-target regression trees and ensembles to model a compound index of vegetation condition. *Ecological Modelling*, 220(8):1159 –1168, 2009.
148. G. Tsoumakas, E. Spyromitros-Xioufis, J. Vilcek, and I. Vlahavas. Mulan: A java library for multi-label learning. *The Journal of Machine Learning Research*, 12:2411–2414, 2011.
149. O. Luaces, J. Díez, J. Barranquero, J. J. del Coz, and A. Bahamonde. Binary relevance efficacy for multilabel classification. *Progress in Artificial Intelligence*, 1(4):303–313, 2012.
150. S. Godbole and S. Sarawagi. Discriminative methods for multi-labeled classification. *Advances in Knowledge Discovery and Data Mining*, pages 22–30, 2004.
151. J. Read, B. Pfahringer, G. Holmes, and E. Frank. Classifier chains for multi-label classification. *Machine Learning*, 85(3):333, 2011.
152. F. Charte, A. J. Rivera, M. J. del Jesus, F. Herrera, and MLeNN. A first approach to heuristic multilabel undersampling. In *Intelligent Data Engineering and Automated Learning – IDEAL 2014*, 15th International Conference, Salamanca, Spain, pages 1–9, 2014.
153. E. S. Xioufis, W. Groves, G. Tsoumakas, and I. P. Vlahavas. Multi-label classification methods for multi-target regression. *CoRR abs/1211.6581.*, 2012.
154. S. Kanj, F. Abdallah, T. Denœux, and K. Tout. Editing training data for multi-label classification with the k-nearest neighbor rule. *Pattern Analysis and Applications*, 19(1):145–161, 2016.
155. Á. Arnaiz-González, J. F. Díez-Pastor, J. J. Rodríguez, and C. I. García-Osorio. Study of data transformation techniques for adapting single-label prototype selection algorithms to multi-label learning. *Expert Systems with Applications*, 109:114–130, 2018.
156. H. Brighton and Ch. Mellish. On the consistency of information filters for lazy learning algorithms. *Springer Berlin Heidelberg*, pages 283–288, 1999.
157. T. Wieczorek. *Neuronowe Modelowanie Procesów Technologicznych*. Silesian University of Technology, 2008.
158. A. Roghani. *Artificial Neural Networks: Applications in Financial Forecasting*. CreateSpace Independent Publishing Platform, 2016.
159. L. Rutkowski. *Computational intelligence: methods and techniques*. Springer, 2008.
160. S. Golak, D. Burchart-Korol, K. Czaplicka-Kolarz, and T. Wieczorek. Application of neural network for the prediction of eco-efficiency. In *ISNN 2011*, volume 6677 of *Part III. LNCS*, page 380–387. Springer, 2011.
161. M. Kordos and A. Cwiok. A new approach to neural network based stock trading strategy. *Lecture Notes in Computer Science*, 6936:429–436, 2011.
162. P. Werbos. Beyond regression: new tools for prediction and analysis in the behavioral science. *Doctoral Dissertation*, 1974.
163. D. E. Rumelhart et al. Learning internal representations by error backpropagation. *Parallel Distributed Processing*, 1:318–362, 1986.
164. W. Duch and N. Jankowski. Survey of neural transfer functions. *Neural Computing Surveys 2*, pages 163–212, 1999.
165. M. Kordos and W. Duch. A survey of factors influencing mlp error surface. *Control and Cybernetics*, 33(4):611–631, 2004.
166. M. Riedmiller and H. Braun. A direct adaptive method for faster backpropagation learning: The rprop algorithm. In *Neural Networks, 1993., IEEE International Conference on*, pages 586–591. IEEE, 1993.
167. M. Kordos and W. Duch. Variable step search algorithm for mlp training. In *the 8th IASTED international Conference on artificial intelligence and soft computing, Marbella, Spain*, pages 215–220, 2004.
168. M. Kordos and W. Duch. Variable step search algorithm for feedforward networks. *Neurocomputing*, 71(13–15):2470–2480, 2008.

169. M. Kordos, A. Rusiecki, T. Kamiński, and K. Greń. Weight update sequence in mlp networks. In *IDEAL 2014*, 2014.
170. D. J. Olive and D. M. Hawkins. Robustifying robust estimators. *Unpublished manuscript*, 2007.
171. K. Liano. Robust error measure for supervised neural network learning with outliers. *IEEE Transactions Neural Netw*, 7(1):246–250, 1996.
172. A. V. Pernia-Espinoza, J. B. Ordieres-Mere, F. J. M. de Pison, and A. Gonzalez-Marcos. Tao-robust backpropagation learning algorithm. *Neural Networks*, 18(2):191–204, 2005.
173. C. C. Chuang, S. F. Su, and C. C. Hsiao. The annealing robust backpropagation(arbp) learning algorithm. *IEEE Transactions Neural Netw*, 11(5):1068–1077, 2000.
174. M. El-Melegy, M. Essai, and A. Ali. Robust training of artificial feedforward neural networks. *Studies in Computational Intelligence*, 201:217–242, 2009.
175. M. El-Melegy. Random sampler m-estimator algorithm for robust function approximation via feed-forward neural networks. In *Neural Networks (IJCNN), The 2011 international joint conference*, pages 3134–3140, 2011.
176. M. El-Melegy. Ransac algorithm with sequential probability ratio test for robust training of feed-forward neural networks. In *Neural Networks (IJCNN), The 2011 international joint conference*, pages 3256–3263, 2011.
177. A. Rusiecki. Robust mcd-based backpropagation learning algorithm. In *ICAISC 2008*, volume 5097 of *LNCS (LNAI)*, pages 154—-163. Springer, 2008.
178. F. Lauer. On the exact minimization of saturated loss functions for robust regression and subspace estimation. *Pattern Recognition Letters*, 112:317–323, 2018.
179. D. Chen and R. Jain. A robust backpropagation learning algorithm for function approximation. *IEEE Transactions Neural Netw*, 5(3):467–479, 1994.
180. A. Rusiecki. Robust learning algorithm based on iterative least median of squares. *Neural Process Lett*, 36(2):145–160, 2012.
181. A. Rusiecki. Robust learning algorithm based on lta estimator. *Neurocomputing*, 120:624–632, 2013.
182. A. Rusiecki. Fault tolerant feedforward neural network with median neuron input function. *Electronics Letters*, 41(10):603—-605, 2005.
183. P. J. Huber. Robust statistics. *Wiley Series in Probability and Statistics*, 1981.
184. P. J. Rousseeuw. Least median of squares regression. *Journal of the American Statistical Association*, 79(388):871–880, 1984.
185. A. Rusiecki. Robust lts backpropagation learning algorithm. *Computational and ambient intelligence*, 4507:102—-109, 2007.
186. J. T. Barron. A general and adaptive robust loss function. *arXiv:1701:03077v6*, 2018.
187. P. Leray and P. Gallinari. Feature selection with neural networks. *Behaviormetrika*, 26:145–166, 1999.
188. S. Bach, A. Binder, G. Montavon, F. Klauschen, K. R. M¨uller, and W. Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one*, 10(7):e0130140, 2015.
189. P.-J. Kindermans, K. Sch¨utt, K.-R. M¨uller, and S. D¨ahne. Investigating the influence of noise and distractors on the interpretation of neural networks. *arXiv preprint*, 2016.
190. A. Datta, S. Sen, and Y. Zick. Algorithmic transparency via quantitative input influence: Theory and experiments with learning systems. In *IEEE Symposium*, Security and Privacy, page 598–617, 2016.
191. J. Zurada, A. Malinowski, and S. Usui. Perturbation method for deleting redundant inputs of perceptron networks. *Neurocomputing*, 14(2):177–193, 1997.
192. X. Zeng and T. R. Martinez. Feature weighting using neural networks. In *IEEE International Joint Conference on Neural Networks IJCNN'4*, pages 327–133, 2004.
193. J. Yoon, J. Jordon, and M. van der Schaar. Invase: Instance-wise variable selection using neural networks. *ICLR 2019*, 2019.

194. A. Solanas, E. Romero, and et. al. Feature selection and outliers detection with genetic algorithms and neural networks. In *Artificial Intelligence Research and Development*, 8th International Conference of the ACIA, pages 26–28, October 2005.
195. A. Shrikumar, P. Greenside, and A. Kundaje. Learning important features through propagating activation differences. *arXiv preprint*, 2017.
196. S. M. Lundberg and S. I. Lee. A unified approach to interpreting model predictions. *Advances in Neural Information Processing Systems*, pages 4765—4774, 2017.
197. W. Duch. Support vector neural training. In *International Conference on Artificial Neural Networks*, volume 3697 of *LNCS*, pages 67–72, 2004.
198. M. Kordos. Data selection for neural networks. *Schedae Informaticae*, 25:153—164, 2017.
199. M. Kordos. Search-based algorithms for multilayer perceptrons. *PhD Thesis*, 2005.
200. G. Towell. Symbolic knowledge and neural networks: Insertion, refinement and extraction. *PhD Thesis*, 1991.
201. L. Fu. Rule generation from neural networks. In *IEEE Transactions on Systems*, volume 28(8) of *Man and Cybernetics*, pages 1114–24, 1994.
202. L. Decloedt, F. Osorio, and B. Amy. Rule-out method: A new approach for knowledge explication from trained ann". *Rule Extraction from Trained Artificial Neural NEtworks Workshop*, pages 34–42, 1996.
203. B. Yoon and R. Lacher. Extracting rules by destructive learning. In *The IEEE Int. Conf. on Neural Networks*, pages 1766–71, 1994.
204. R. Setiono. Extracting m-of-n rules from trained neural networks. In *IEEE Transactions on Neural Networks*, volume 11, pages 512–519, 2000.
205. A. Gupta, S. Park, and S. M. Lam. Generalized analytic rule extraction for feedforward neural networks. In *IEEE Transactions on Knowledge and Data Engineering*, volume 11, 1999.
206. R. Setiono, B. Baesens, and C. Mues. A note on knowledge discovery using neural networks and its application to credit card screening. *European Journal of Operational Research*, 192:326–332, 2008.
207. R. Andrews and S. Geva. Rule extraction from a constrained error backpropagation ml. In *5th Aust. Conf. Neural Networks*, Procedia Computer Science, 1994.
208. R. Setiono and W. K. Leow. Fernn: An algorithm for fast extraction of rules from neural networks. *Applied Intelligence*, 12:15–25, 2000.
209. L. Ozbakir, A. Baykasoglu, and S. Kulluk. A soft computingbased approach for integrated training and rule extraction from artificial neural networks. *Applied Soft Computing*, 10(1):304–317, 2010.
210. D. Kim and J. Lee. Handling continuous-valued attributes in decision tree with neural network modeling. In *11th european conference on machine learning*, volume 1810 of *Lecture notes in computer science*, pages 211–219, 2000.
211. R. Krishnan, G. Sivakumar, and P. Bhattacharya. A search technique for rule extraction from trained artificial neural networks. *Pattern recognition letters*, 20:273–280, 1999.
212. R. Setiono, W. K. Leow, and J. M. Żurada. Extraction of rules from artificial neural networks for nonlinear regression. In *IEEE Transactions on Neural Networks*, volume 13, May 2002.
213. S. Thrunm. Extracting rules from artificial neural networks with distributed representation. *Advances in Neural Information Processing Systems*, 7, 1995.
214. U. Markowska-Kaczmar and M. Chumieja. Rule extraction from neural networks with evolutionary algorithms. In *6th Int. Conf. on Artificial Intelligence and Soft Computing (ICAISC)*, pages 370–37, 2002.
215. U. Markowska-Kaczmar and P. Wnuk-Lipiński. Rule extraction from neural networks by genetic algorithms with pareto optimization. In *7th Int. Conf. on Artificial Intelligence and Soft Computing (ICAISC)*, pages 450–455, June 2004.

In machine learning performance of predictive models is limited by the quality of training data. Feature and instance selection can be used to improve this quality by removing noisy and redundant data. This improves the model prediction, accelerates its learning and makes the data easier to analyze and interpret.

The book comprises three groups of instance selection methods: similarity-based, evolutionary and embedded. In each group several solutions developed by the author and by other researchers are presented. Additionally instance weighting and joint instance and feature selection is discussed.

Similarity-based instance selection methods decide upon the instance selection or rejection, based on how this instance fits into its neighborhood. The advantage of this group is simplicity of implementation and possibility to get an explanation why particular instances were selected.

In evolutionary instance selection methods instances are usually encoded into chromosomes. The selection process minimizes two objectives: data size and error of the predictive model trained on the selected instances. Moreover, multi-objective algorithms generate a pool of solutions with different trade-offs between the objectives. The advantage of this group is high prediction accuracy and strong data reduction of the obtained solutions.

Embedded instance selection methods are incorporated into the predictive model learning process. The book focuses on neural network models. The advantage of this group is that a separate step of instance selection is not required, the obtained results are optimized for that particular predictive model and in many cases we can also get an explanation why given instances were selected.

ISBN 978-83-66249-11-0
ISSN 1643-983X