26.01.2025

Projekt - przedmiot UMA

Zespół w składzie:
Jakub Bąba
Adrian Murawski

Treść zadania

Przygotować implementację algorytmu do indukcji reguł [IRep++] oraz przeprowadzić eksperymenty na wybranym zadaniu z bazy.

Streszczenie założeń

Zadanie ma na celu implementację algorytmu IRep++ oraz przeprowadzenie eksperymentów, które pokażą działanie i wyniki tego algorytmu. Implementacja wykonana została w Pythonie. Do eksperymentów używamy zbioru dotyczącego klasyfikacji raka piersi - jako złośliwego, bądź też łagodnego.

Pełen opis funkcjonalny i struktura projektu

W katalogu *src* znajdują się pliki *irep.py, irepplusplus.py i ripper.py*, które zawierają implementację wspomnianych wyżej algorytmów. Najważniejsze metody klas implementujących algorytmy, to *fit()* i *predict()*. Pierwsza z nich uruchamia cały proces nauki reguł przez algorytm - na podstawie przekazanych do metody danych treningowych. Druga pozwala na wygenerowanie przewidywanych klas dla przekazanych danych testowych na podstawie wytrenowanego modelu.
Dodatkowo, w każdym algorytmie można zmodyfikować stosunek zbioru tworzącego do ograniczającego, a także wpłynąć na maksymalną ilość iteracji w głównej pętli w *fit()* oraz pętli tworzenia reguły w *learn_rule()*.
W katalogu src znajdują się także pliki:
prepare_data.py - funkcje związane z przygotowywaniem plików .csv i konwertowaniem ich do DataFrame;
train_and_predict.py - wykorzystanie algorytmów, funkcje które po przyjęciu danych i odpowiednich parametrów przechodzą przez proces trenowania oraz oceniania danych;
experiments.py - wykonanie wszystkich eksperymentów opisanych poniżej i zapisanie wyników do plików;
plot_results.py - wczytanie danych wynikowych z plików i przygotowanie wykresów.

Poza folderem src, w repozytorium znajdują się foldery:
data - w podfolderze raw znajduje się oryginalny plik z danymi z platformy Kaggle, a w podfolderze processed znajduje się plik z dostosowanymi kolumnami, a także pliki ze zmodyfikowanymi danymi (szczegóły w eksperymentach);
results/accuracy - wyniki eksperymentów w formacie json;
plots - konwersja danych do wykresów.

Opis algorytmu

Algorytm IREP++ jest algorytmem indukcji reguł decyzyjnych. Buduje on reguły decyzyjne oraz przycina je bazując na estymacji błędu rzeczywistego (na podstawie zbioru walidacyjnego). Algorytm ten bazuje swoje działanie na algorytmach RIPPER oraz IREP (Induced Reduced Error Pruning), jednak usprawnia proces tworzenia reguł, dbając o ich odpowiednią ogólność oraz minimalizując ryzyko nadmiernego dopasowania.

Zaprojektowany jest on do klasyfikacji binarnej, czyli do używania na danych, których funkcja celu przyjmuje dwie wartości.

Algorytm działa następująco:

1. Dzieli zbiór treningowy na dwie części:
    - grow set - wykorzystywany do tworzenia reguł,
    - prune set - wykorzystywany do przycinania reguł.
2. Tworzy pustą regułę, ustawiającą wartość funkcji celu na jedną z wartości.
3. Wykonuje kroki aż wszystkie przypadki z grow set zostaną sklasyfikowane lub nie będzie można tworzyć nowych reguł:
    a. Buduje reguły - uszczegóławia i rozszerza istniejącą regułę o dodatkowe warunki bazując na zbiorze grow set.
    b. Przycina reguły - sprawdza nową regułę na zbiorze prune set oraz usuwa te warunki, które nie poprawiają wydajności, pozbywając się nadmiernego dopasowania.
    c. Usuwa przykłady spełniające warunki z grow set (w celu skupienia się na pozostałych przypadkach).

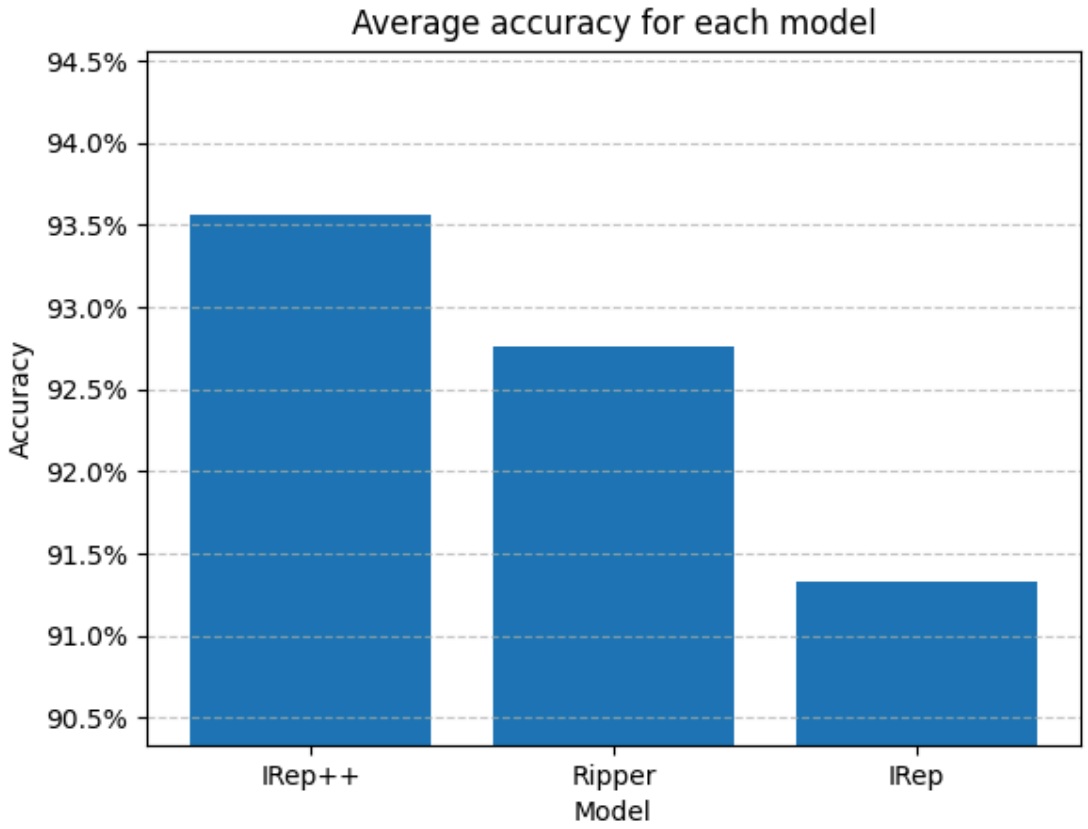
Algorytm tworzy kolejne reguły do momentu spełnienia warunku stopu.

W testach do porównania użyto również implementacji algorytmu IRep oraz Ripper, które są poprzednikami IRep++, ale ze względu na to, że głównym celem projektu była implementacja IRep++, to te algorytmy nie będą opisywane w tym sprawozdaniu.

Raport z przeprowadzonych testów i wnioski

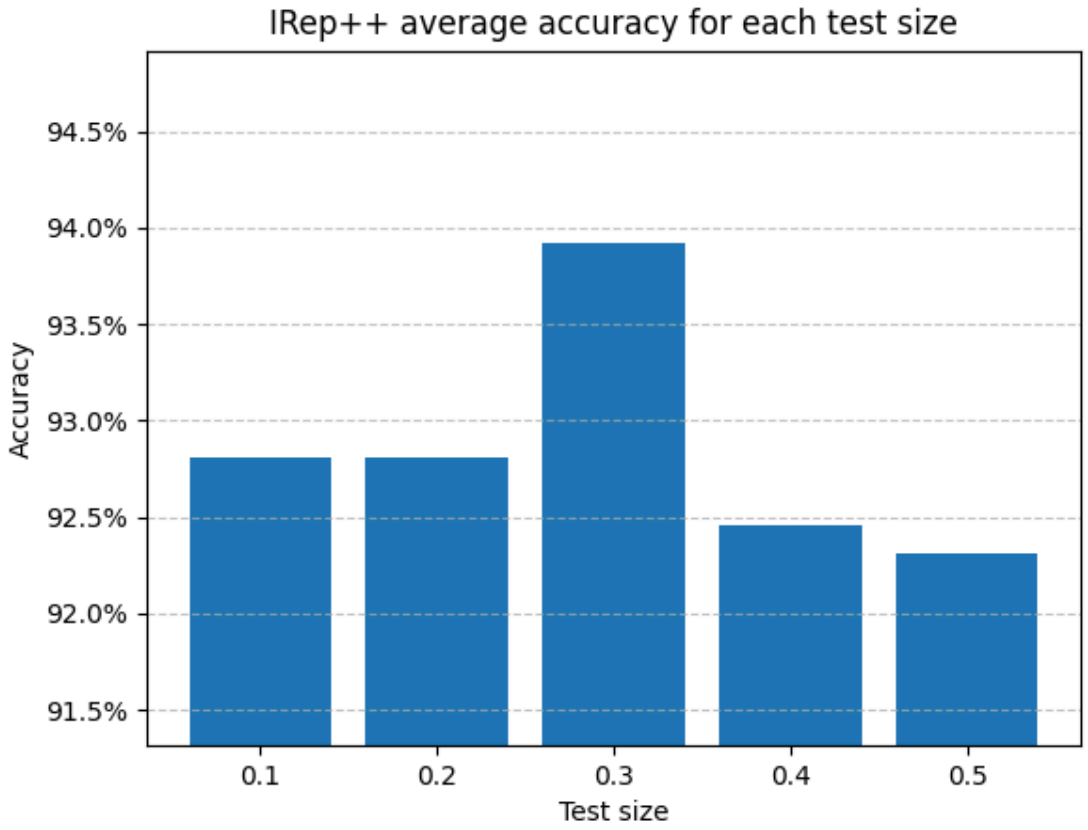
Wszystkie testy zostały przeprowadzone na bazie średniej z dziesięciokrotnego uruchomienia, aby zminimalizować wpływ losowości.

Porównanie algorytmów IREP++, RIPPER oraz IREP na zbiorze danych



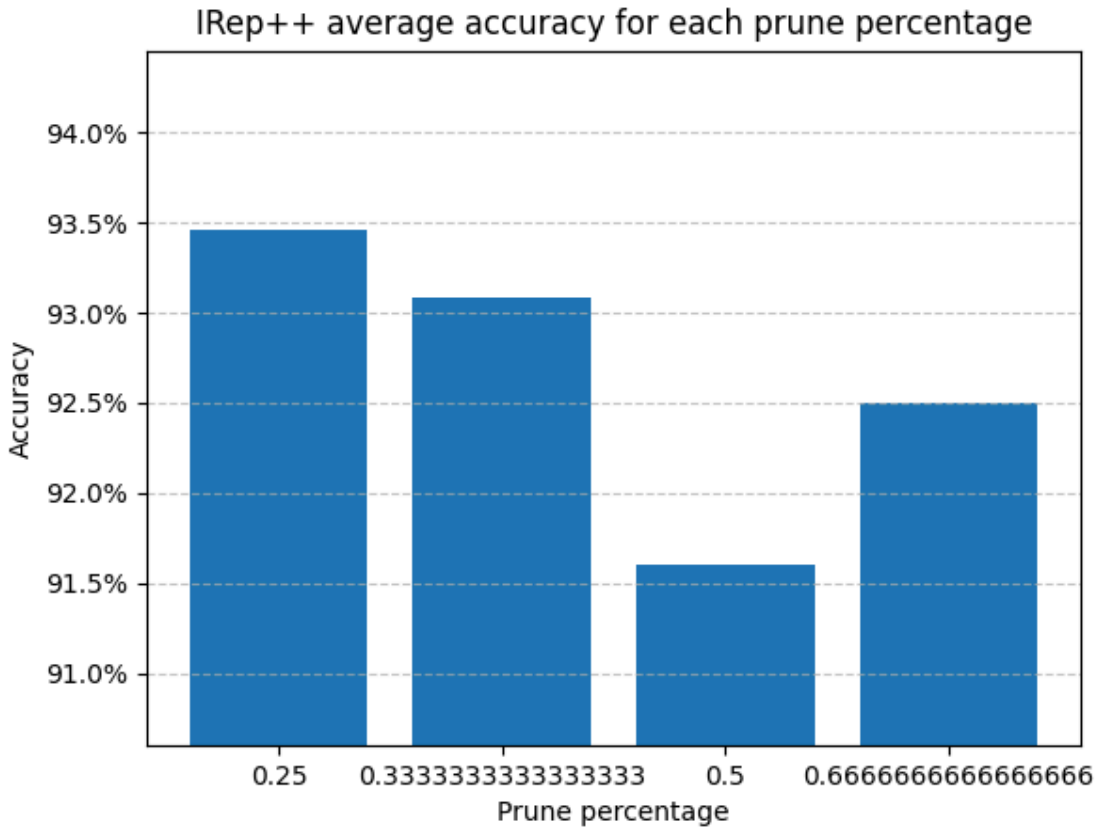
W pierwszej kolejności zdecydowaliśmy się na porównanie dokładności modeli. Można z niego wywnioskować, że najlepiej poradził sobie algorytm IRep++ (93.5%), ale różnica pomiędzy nim, a RIPPER (92.75%) jest niewielka. Zdecydowanie najgorzej poradził sobie algorytm IRep (91.3%).
Jest to zgodne z koncepcją algorytmów - RIPPER miał być ulepszeniem algorytmu IRep, a IRep++ miał w założeniu radzić sobie podobnie do algorytmu RIPPER, będąc jednocześnie mniej złożonym.

Modyfikacja rozmiaru zbioru treningowego i wpływ takiej modyfikacji na dokładność



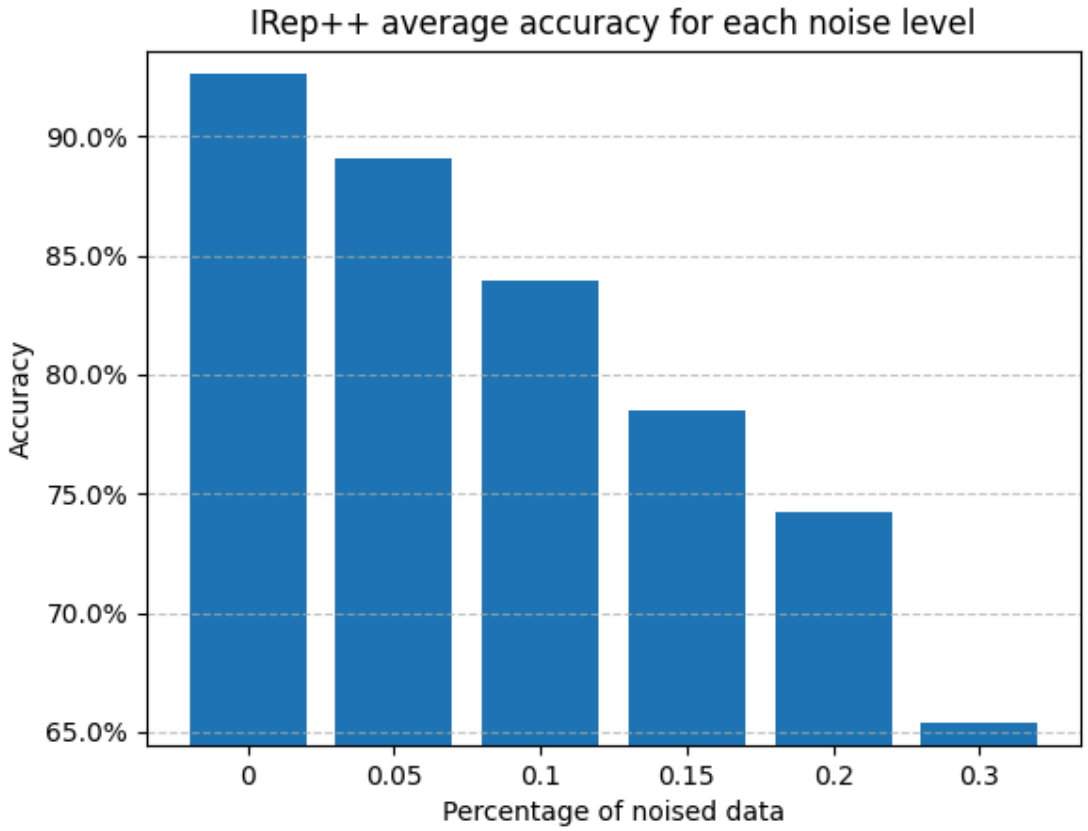
Zbadaliśmy następnie wpływ rozmiaru stosunków zbioru treningowego do zbioru testowego. Algorytm osiągał najlepsze wyniki, gdy zbiory treningowe były podzielone w stosunku 70/30. Większa ilość danych treningowych powodowała trochę gorsze wyniki, ale jeszcze gorsze były one, gdy danych testowych było więcej (60/40, 50/50).
Niższe wyniki przy większej ilości danych testowych były spowodowane pewnym niedotrenowaniem, natomiast w wypadku większej ilości treningowych obniżenie średniego wyniku mogło wynikać z większego błędu statystycznego (stosunkowo mała ilość testów) oraz potencjalnie z przetrenowania modelu. Średnia dokładność modelu jest znacznie większa w przypadku podziału 70/30 - można wnioskować, że dla takiego podziału algorytm jest dobrze wyważony między odpowiednim wytrenowaniem do przewidywania klasy dla danego przykładu, ale jednocześnie nie jest przetrenowany - zbyt dopasowany do przykładów trenujących.

Modyfikacja stosunku rozmiaru zbioru budującego drzewo, do tego je ograniczającego (domyślnie jest to ⅔ do ⅓) i sprawdzenie dokładności



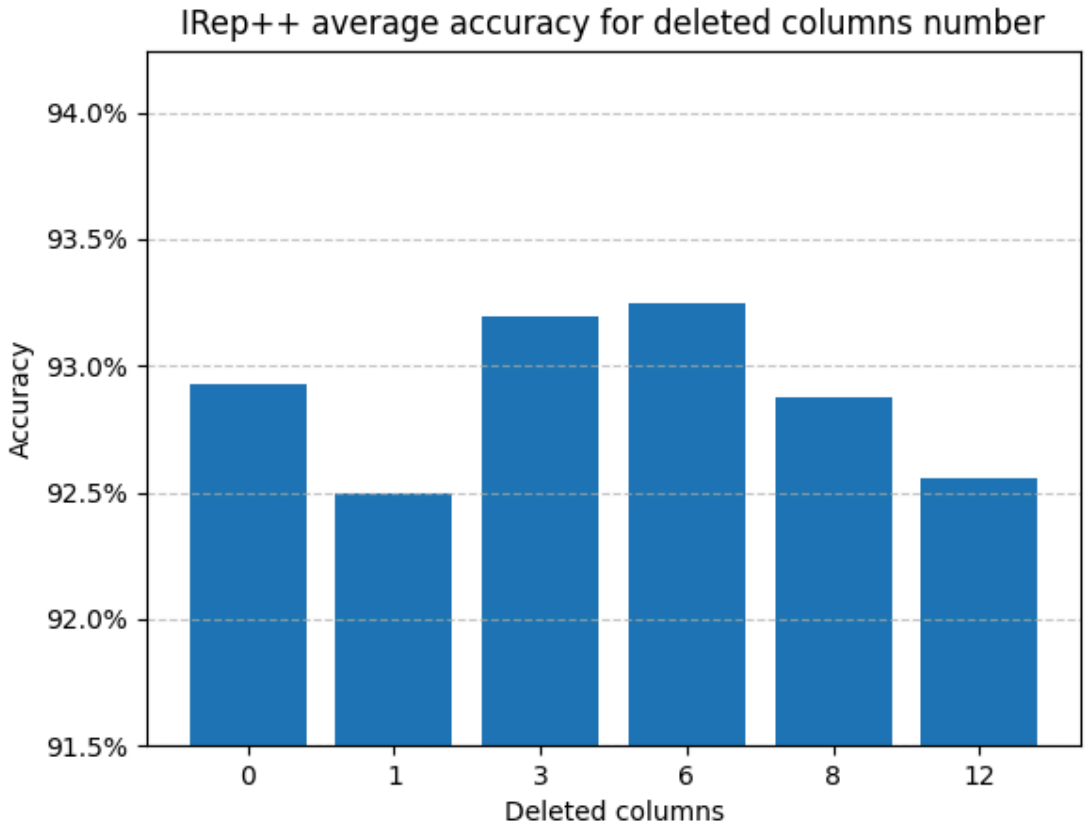
Sprawdziliśmy wpływ stosunku zbioru budującego drzewo do tego je ograniczającego i zauważyliśmy, że algorytm generalnie radził sobie lepiej, gdy rozmiar danych budujących był większy.

Odporność algorytmu na szum



Zdecydowaliśmy się także na zmodyfikowanie danych wejściowych poprzez wprowadzenie do nich szumów - modyfikację pewnego procenta parametrów o maksymalnie 10%. Zauważyliśmy dzięki temu, że z małym zaszumieniem (rzędu 5% komórek) algorytm radził sobie nieźle, natomiast im większe zaszumienie, tym gorsza dokładność, sięgająca jedynie 65% przy trzydziestoprocentowym zaszumieniu.

Przydatność poszczególnych kolumn do ostatecznego wyniku



Dane mają 30 kolumn. W związku z tym sprawdziliśmy które były najczęściej wykorzystywane w regułach, a następnie usunęliśmy najczęściej występujące. Zauważyliśmy, że w regułach występowały najczęściej trzy kolumny - w związku z tym w pierwszej kolejności usunęliśmy jedną kolumnę która występowała najczęściej w regułach, a następnie pozostałe dwie. Przy usunięciu jednej z tych kolumn okazało się, że wyniki były zdecydowanie gorsze niż przy usunięciu trzech. Bardzo możliwe, że z racji że te trzy reguły często występowały wspólnie, bez jednej z nich pozostałe zaczęły mieć zbyt duży wpływ. Usuwaliśmy dalej po kilka często występujących kolumn, aż do 12. Okazało się, że przy usunięciu 3 i 6 kolumn wyniki były nawet lepsze niż bazowe - może to wynikać z nadmiarowej ilości informacji, możliwe że aż 30 kolumn powodowało nadmierne dopasowania.
Po usuwaniu kolejnych często występujących kolumn zauważyliśmy że wyniki zaczęły ponownie spadać. Jest to spowodowane tym, że pozostałe kolumny były dużo rzadziej wykorzystywane. Wciąż wyniki plasowały się jednak powyżej 92% - duża ilość kolumn w naszych danych powodowała, że niektóre dane mogły być ze sobą wzajemnie powiązane.

Opis wykorzystanych narzędzi, bibliotek

Do całości implementacji wykorzystaliśmy Pythona. Narzędzia i biblioteki, z których skorzystaliśmy to ***pandas*** do przechowywania danych w wygodnej formie oraz ***numpy*** do łatwych obliczeń na zbiorach. Dodatkowo wykorzystaliśmy pomocnicze funkcje *accuracy_score* i *train_test_split* z biblioteki ***sklearn***.

Źródła

https://www.researchgate.net/publication/220907085_IREP_a_Faster_Rule_Learning_Algorithm
http://elektron.elka.pw.edu.pl/~pcichosz/uma/slajdy/uma-s9.pdf
https://www.kaggle.com/datasets/yasserh/breast-cancer-dataset