# Projekt ALHE - dokumentacja końcowa

Adam Szałowski Daniel Chmielewiec

16.06.2021

## Treść zadania i opis problemu

13. Optymalizacja parametrów algorytmu xgboost z wykorzystaniem algorytmów ewolucyjnych / genetycznych. Zadanie polega na wybraniu ciekawego zbioru do zagadnienia klasyfikacyjnego oraz wykorzystanie go do predykcji za pomoca algorytmu xgboost. Główna częścią zadania będzie optymalizacja hiperparametrów (algorytmu xgboost) z wykorzystaniem algorytmu ewolucyjnego / genetycznego.

Algorytm XGBoost posiada wiele hiperparametrów, których zmiana wpływa na skuteczność klasyfikacji nauczonego modelu. Ich domyślne wartości często prowadzą do zadowalających efektów, jednak odpowiednia manipulacja nimi może zwiększyć skuteczność. Parametrów jest na tyle dużo i są na tyle różnorodne, że ich arbitralne dobieranie jest trudne i nieefektywne.

## Przyjęte założenia

### Środowisko

Zadanie wykonane będzie przy użyciu języka Python3. Wykorzystamy implementacje algorytmu xgboost z biblioteki xgboost.

### Parametry

Opis wszystkich parametrów, które przyjmuje wybrana implementacja algorytmu xgboost, można znaleźć tutaj. Wybraliśmy następujące 6 parametrów, które algorytm genetyczny będzie starał się dobrać: `eta, gamma, min_child_weight, max_delta_step, subsample, scale_pos_weight` .

### Funkcja oceny

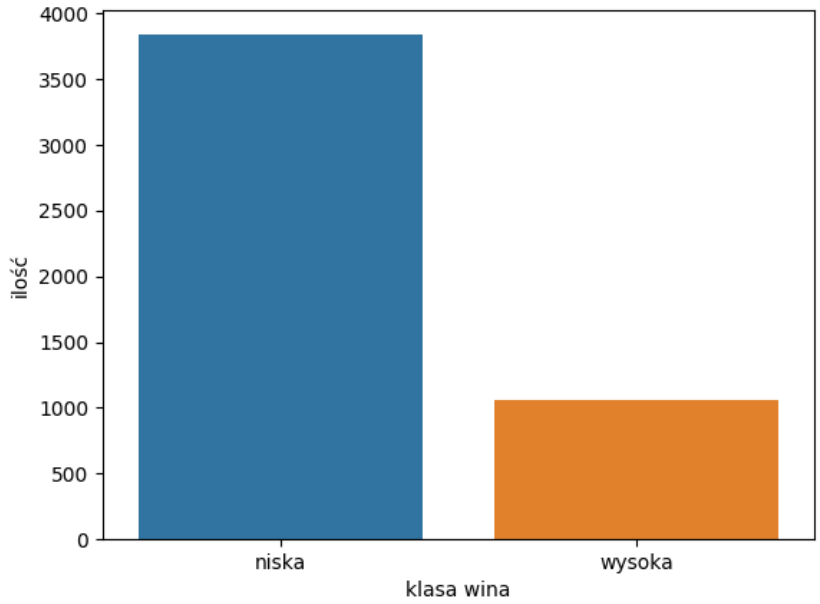
W przypadku naszego zadania jako miarę jakości danego osobnika przyjmujemy wartość funkcji F1-score liczoną według następującego wzoru:

$$F_1 = \frac{TP}{TP + \frac{1}{2}(FP + FN)}$$

gdzie $TP$ - true positives; $FP$ - false positives; $FN$ - false negatives.

### Wybrany zbiór danych

Wybraliśmy zbiór zawierający informację na temat białych win ze strony UCI Machine Learning Repository. Problem polega na klasyfikacji wina na podstawie 11 atrybutów na jedną z klas oznaczających jakość ( wartości pomiędzy 0 i 10 ). Zdecydowaliśmy się zbinaryzowanie zestawu klas dzieląc wina na niską i wysoką. Wino jest wysokiej klasy jeśli jego jakość jest większa lub równa 7, a niskej klasy w przeciwnym wypadku. Poniższy wykres przedstawia rozkład tych klas w zbiorze:




Istotnym aspektem zbioru jest niezbalansowanie klas co oznacza, że ważne jest odpowiednie ustawienie parametrów `subsample` oraz `scale_pos_weight`, w czym pomóc ma algorytm genetyczny.

## Opis rozwiązania

Metody selekcji, krzyżowania oraz mutacji, a także parametry do nich zostały wybrane na drodze przeprowadzania testów i ich analizy. Ustalone wartości i metody wydawały się dawać najlepsze rezultaty. Wielkość populacji została ustalona na 100 osobników. Pojedynczy osobnik jest zbiorem parametrów, które będą optymalizowane.

By uniknąć przeuczenia dla każdej generacji zbiór danych losowo dzielony jest na

część trenującą i testującą, domyślnie część trenująca stanowi 80%. Na początku generowana zostaje populacja startowa i poddawana jest ocenie przy użyciu tych zbiorów. Parametry dla generowanych osobników są losowane. Na podstawie miary jakości osobników, algorytm bedzie generował kolejne zestawy według typowego ewolucyjnego postępowania:

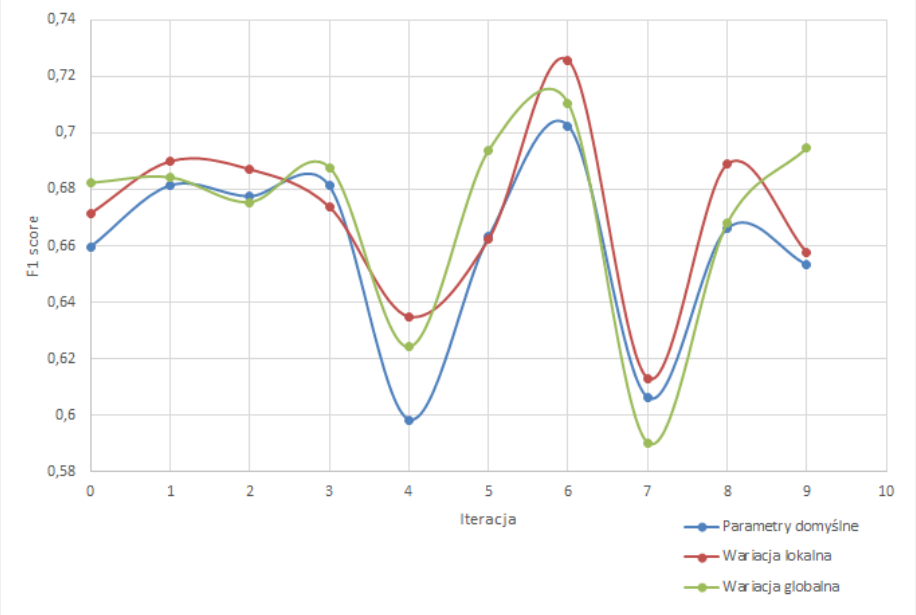
- Selekcja - zastosowaliśmy selekcję rangową, która polega na przypisaniu prawdopodobieństwa każdemu osobnikowi z populacji na podstawie jego rangi. Ranga osobnika to jego pozycja w posortowanej rosnąco liście zawierającej wszystkie osobniki z populacji. Prawdopodobieństwo wyboru osobnika to jego ranga podzielona przez sumę wszystkich rang.
- Krzyżowanie - użyliśmy krzyżowania równomiernego z parametrem równym 0.5
- Mutacja - dla każdego parametru osobnika ustalone jest prawdopodobieństwo mutacji (parametr `mutation_prob`). Zaimplementowane zostały dwa operatory mutacji:
  - Mutacja lokalna polega na wylosowaniu wartości z rozkładu normalnego o odpowiednio dobranej dla każdego parametru wariancji i dodaniu jej do aktualnej wartości. W tym przypadku zastosowaliśmy sprawdzanie ograniczeń i naprawę przez rzutowanie.
  - Mutacja globalna zastępuje wartość parametru wartością wylosowaną z rozkładem jednorodnym z całego przedziału parametru
- Sukcesja - generowanie nowej populacji polega na selekcji dwóch osobników, które następnie są krzyżowane a ich potomek zostaje dodany do nowej populacji z pewnym prawdopodobieństwem (parametr `crossover_prob`) lub w przeciwnym wypadku selekcji jednego osobnika i dodania go do nowej populacji
- Ocena - wszystkie osobniki zostają poddane ocenie poprzez nauczenie na ich parametrach klasyfikatora, a następnie obliczeniu funkcji celu na podstawie zbioru testowego. W każdej iteracji zbiór testujący i uczący jest losowany oddzielnie, aby uniknąć przeuczenia.

Opisany proces wykonuje się przez podaną liczbę iteracji - domyślnie 10.

## Przeprowadzone eksperymenty i analiza wyników

### Porównanie metod mutacji z domyślnymi parametrami

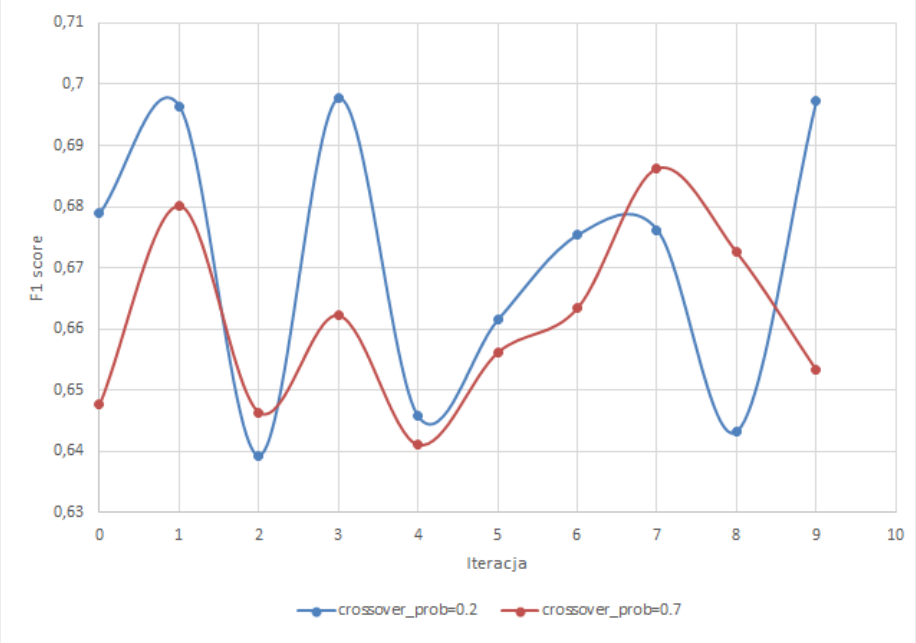
Przeprowadzone testy odbywały się poprzez uruchomienie algorytmu dla mutacji globalnej oraz lokalnej tak, że dla każdej iteracji po przejściu przez 10 generacji zwracany był najlepszy osobnik z ostatniego pokolenia i poddawany testowi na nowym podziale zbiór uczący-testowy i porównany z klasyfikatorem z domyślnymi parametrami.




Dane na wykresie pokazują, że w większości przypadków osobnik wygenerowany przez algorytm radził sobie lepiej z predykcją na nowym zbiorze testowym, niż domyślny zestaw parametrów. Wynik zastosowania wariacji lokalnej jak i globalnej jest niższy od wyniku dla domyślnych wartości parametrów 2 razy, przy czym po razie jedynie nieznacznie.

### Porównanie dla różnych wartości parametru `crossover_prob`

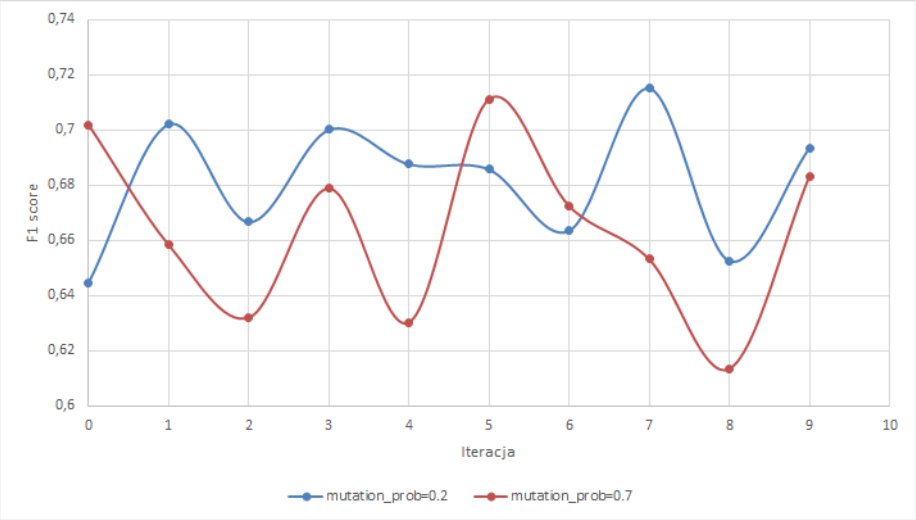
Dokonaliśmy porównania wyników uzyskanych przez algorytm dla różnych wartości `crossover_prob` przy zachowanym parametrze `mutation_prob=0.5` oraz mutacji globalnej. W każdej iteracji algorytmy uczone są na zbiorze treningowym, a następnie obliczana jest wartość funkcji *F1-score* dla predykcji na zbiorze testowym dla najlepszego osobnika po 10 generacjach.




W większości przypadków mniejsze prawdopodobieństwo krzyżowania daje lepsze rezultaty, jednak jest to niewielka przewaga - 6 na 10 przypadków.

### Porównanie dla różnych wartości parametru `mutation_prob`

Dokonaliśmy porównania wyników uzyskanych przez algorytm dla różnych wartości `mutation_prob` przy zachowanym parametrze `crossover_prob=0.3` oraz mutacji globalnej. W każdej iteracji algorytmy uczone są na zbiorze treningowym, a następnie obliczany jest wartość funkcji *F1-score* dla predykcji na zbiorze testowym dla najlepszego osobnika po 10 generacjach.




Na wykresie możemy zauważyć, że dla niższego prawdopodobieństwa mutacji algorytm średnio osiągał lepsze wyniki ale różnica ta nie jest bardzo znacząca.

## Wnioski

Dobieranie hiperparametrów przy pomocy algorytmu genetycznego dla algorytmu XGBoost okazało się trudnym zadaniem. Ostatecznie udało nam się nieznacznie poprawić jego działanie ale wydaje nam się, że w tym wypadku lepiej byłoby zastosować inne podejścia uczenia maszynowego.

Dużą przeszkodą był tutaj bardzo długi czas działania algorytmu genetycznego nawet dla niewielkich ilości generacji ponieważ obliczanie funkcji celu dla każdego osobnika jest bardzo kosztowne czasowo. Uniemożliwiło to wygenerowanie większej ilości generacji lub zaimplementowania walidacji krzyżowej wewnątrz algorytmu genetycznego ponieważ algorytm działałby zbyt długo. Rozwiązaniem tego mogłoby na przykład być zaimplementowanie rozproszonego uczenia na wielu maszynach lub wykorzystanie serwerów chmurowych do zwiększenia mocy obliczeniowej.

## Instalacja i uruchomienie

Kod źródłowy projektu znajduje się w repozytorium GitLab. Wymagane moduły zapisane są w pliku *requirements.txt* i można zainstalować je przy użyciu komendy `pip install -r requirements.txt`

Przykład optymalizacji przy użyciu algorytmu genetycznego:

```
from XGBoostTuner import XGBoostTuner
from preprocessing import processWineData

filename = '../data/winequality-white.csv'
X, y = processWineData(filename)

xgb_tuner = XGBoostTuner(X, y)
xgb_tuner.run()
params = xgb_tuner.getBestParams()
```

Skrypty z eksperymentami znajdują się w pliku *src/main.py*.