Przeszukiwanie i optymalizacja

Dokumentacja końcowa

Skład zespołu:

Ignacy Dąbkowski
Ireneusz Okniński

Treść zadania

Optymalizacja hiperparametrów algorytmu xgboost za pomocą algorytmów heurystycznych

Algorytm xgboost jest uważany za jedną z najlepszych metod klasyfikacji. Niestety jego stosowanie jest utrudnione ze względu na bardzo dużą liczbę hiperparametrów takich jak np. liczba produkowanych drzew decyzyjnych, ich głębokość, krok uczenia, wielkość podzbiorów losowanych przy kolejnym kroku etc. Wymienione atrybuty często optymalizowane są ręcznie w ograniczony sposób. Zadanie to można jednak robić w mądrzejszy sposób wykorzystując algorytmy heurystyczne. W ramach tematu tego projektu należy przetestować różne algorytmy heurystyczne/populacyjne w kontekście problemu strojenia hiperparametrów algorytmu xgboost. Jakość działania algorytmów należy przetestować na podstawie dowolnego (nietrywialnego) zbioru. Np: https://www.kaggle.com/c/porto-seguro-safe-driver-prediction.

Implementacja

utils.py

Funkcja `eval_gini` oblicza współczynnik Gini między rzeczywistymi etykietami (`y_true`) a przewidywanymi wartościami (`y_pred`). Współczynnik Gini to miara nierówności, często używana w kontekście oceny jakości modeli predykcyjnych.

params.py

Definiuje strukturę do zarządzania parametrami modelu XGBoost, umożliwiającą generowanie, mutację i serializację różnych typów parametrów.

- **XGBoostParam (Klasa abstrakcyjna)**
    - Jest klasą abstrakcyjną, definiującą interfejs dla różnych typów parametrów XGBoost.

    - Zawiera metody abstrakcyjne `serialize`, `generate_random_param` i `get_mutated_point`, które muszą być zaimplementowane przez konkretne podklasy.
- **ContinuousParam**
    - Reprezentuje parametr XGBoost przyjmujący ciągłe wartości.
    - Przechowuje zakres (`min_value` i `max_value`) dla tego parametru.
    - Implementuje metody do serializacji, generowania losowej wartości oraz mutacji wartości dla tego typu parametru.
- **BinaryParam**
    - Reprezentuje parametr XGBoost, który przyjmuje wartości binarne (0 lub 1).
    - Implementuje metody do serializacji, generowania losowej wartości oraz mutacji wartości dla tego typu parametru.
- **CategoricalParam**
    - Reprezentuje parametr XGBoost, który przyjmuje wartości kategoryczne zdefiniowane w postaci słownika.
    - Implementuje metody do serializacji, generowania losowej wartości oraz mutacji wartości dla tego typu parametru.
- **DiscreteParam**
    - Reprezentuje parametr XGBoost przyjmujący dyskretne wartości z określonego zakresu.
    - Przechowuje minimalną i maksymalną wartość dla tego parametru.
    - Implementuje metody do serializacji, generowania losowej wartości oraz mutacji wartości dla tego typu parametru.
- **params_factory**
    - Funkcja fabryki, która na podstawie przekazanego rodzaju parametru tworzy instancję odpowiedniej klasy parametru.
    - Obsługuje typy parametrów: 'continuous', 'binary', 'categorical', 'discrete'. Jeśli rodzaj nie jest obsługiwany, zgłasza błąd.

config.py

Definiuje strukturę do zarządzania parametrami modelu XGBoost, a także dostarcza predefiniowane zestawy parametrów dla różnych rodzajów boosterów (drzewiastego i liniowego) oraz konfiguracje drzewa i liniowe.

- **Boosters**
    - Tuple zawierające dostępne rodzaje boosterów: "gbtree" (drzewiasty) i "gblinear" (liniowy).
- **TREE_PARAMS**
    - Lista zawierająca predefiniowane konfiguracje parametrów dla modelu drzewiastego XGBoost.
    - Każdy element listy to słownik z informacjami o parametrze, takimi jak nazwa, typ, oraz opcjonalne minimalne i maksymalne wartości lub kategorie dla parametrów ciągłych lub dyskretnych.
- **LINEAR_PARAMS**
    - Lista zawierająca predefiniowane konfiguracje parametrów dla modelu liniowego XGBoost.

    - Każdy element listy to słownik z informacjami o parametrze, takimi jak nazwa, typ i ewentualne kategorie dla parametrów ciągłych lub kategorycznych.

point.py

Definiuje klasy `Booster` i `Point`, które są używane do zarządzania parametrami modelu XGBoost w kontekście algorytmu optymalizacji heurystycznej.

- **Booster:**
    - Klasa reprezentująca rodzaj boostera w modelu XGBoost.
    - Posiada atrybuty `_name` (nazwa boostera) i `_params` (lista parametrów tego boostera).
    - Udostępnia właściwość `name` do odczytu i zapisu, zapewniając sprawdzenie poprawności boostera.
    - Udostępnia właściwości `length` (długość listy parametrów) i `params` (parametry boostera).
- **Point:**
    - Klasa reprezentująca punkt w przestrzeni poszukiwań parametrów modelu XGBoost.
    - Posiada atrybuty `_boosters` (krotka zawierająca dwa obiekty Booster) i `_current_booster` (obiekt Booster wybrany jako bieżący).
    - Udostępnia właściwość `params` (wartości parametrów dla obecnego boostera) oraz metodę `swap_current_booster` do zamiany bieżącego boostera.
    - Implementuje metody `neighbour` i `mutate` do generowania nowych punktów poprzez zamianę boostera lub mutację parametrów.
    - Udostępnia metodę `serialize` do konwersji punktu na słownik z parametrami XGBoost.
- **generate_random_point:**
    - Funkcja generująca losowy punkt w przestrzeni parametrów modelu XGBoost.
    - Tworzy boostery dla modelu drzewiastego i liniowego, a następnie inicjalizuje punkt z jednym z tych boosterów jako bieżący.
    - Losowo decyduje, czy bieżącym boosterem ma być model drzewiasty czy liniowy.

Algorytmy

algorithm.py

Ten kod dostarcza struktury bazowej dla implementacji konkretnych algorytmów optymalizacji heurystycznej, które mają optymalizować parametry modelu XGBoost.

- **Algorithm (Klasa abstrakcyjna):**
    - Klasa abstrakcyjna definiująca interfejs dla algorytmów optymalizacji heurystycznej.

    - Posiada atrybuty przechowujące informacje o bieżącym modelu (`_current_booster`), zbiorze testowym (`_y_test`, `_train_matrix`, `_test_matrix`), liczbie rund optymalizacji (`_num_boost_rounds`), pliku wyjściowym (`_output_file`) oraz metrykach i parametrach modelu.
    - Posiada atrybuty do śledzenia najlepszego punktu (`_best_point`), jakości (`_best_quality`) oraz historii jakości (`_quality_history`).
    - Definiuje abstrakcyjne metody `run` i `init_params`, które muszą być zaimplementowane przez konkretne algorytmy.
- **run:**
    - Abstrakcyjna metoda, która ma być zaimplementowana przez konkretne algorytmy.
    - Odpowiada za przeprowadzenie procesu optymalizacji przez zadaną liczbę rund (`n_rounds`).
    - Zwraca krotkę zawierającą najlepszy punkt, jakość oraz historię jakości.
- **init_params:**
    - Abstrakcyjna metoda do inicjalizacji parametrów algorytmu na podstawie przekazanego słownika `params`.
    - Przygotowuje macierze danych treningowych i testowych, ustawia parametry algorytmu oraz opcjonalnie wczytuje informacje o pliku wyjściowym, liczbie rund, metrykach i innych.
- **_log:**
    - Metoda do logowania informacji o przebiegu optymalizacji, takich jak numer rundy, najlepsza jakość i najlepszy punkt.
    - Wypisuje informacje na konsolę lub zapisuje je do pliku, jeśli podany jest `_output_file`.
- **_calculate_quality:**
    - Metoda prywatna do obliczania jakości modelu na podstawie przekazanego punktu.
    - Tworzy model XGBoost na podstawie parametrów punktu i trenuje go na danych treningowych.
    - Na końcu ocenia jakość modelu na danych testowych i zwraca wynik.
    - Jakość może być oceniana za pomocą metryki "accuracy" lub "gini", zależnie od ustawień.

RS.py (Przeszukiwanie losowe)

Klasa `RS` (Random Search) implementuje algorytm optymalizacji heurystycznej, który przeszukuje przestrzeń parametrów modelu XGBoost, generując losowe punkty i oceniając ich jakość.

1. **RS (Random Search):**
    - Klasa dziedzicząca po klasie abstrakcyjnej `Algorithm`.
    - Implementuje konkretny algorytm Random Search.
    - **run:**
    - Metoda `run` przeprowadza optymalizację przez zadaną liczbę rund (`n_rounds`).

    - W każdej rundzie generuje losowy punkt, oblicza jego jakość i porównuje z dotychczasowym najlepszym punktem.
    - Aktualizuje najlepszy punkt, jeśli nowo wygenerowany ma lepszą jakość.
    - Opcjonalnie loguje postęp optymalizacji na konsolę.
    - Zwraca krotkę zawierającą najlepszy punkt, jego jakość oraz historię jakości w kolejnych rundach.
    - **init_params:**
    - Metoda inicjalizująca parametry algorytmu na podstawie przekazanego słownika `params`.
    - Wywołuje metodę `init_params` z klasy nadrzędnej do ustawienia podstawowych parametrów.
    - **_random_point:**
    - Metoda prywatna, generująca losowy punkt w przestrzeni parametrów modelu XGBoost.
    - Wykorzystuje funkcję `generate_random_point` z modułu `point`.

SA.py (Symulowane wyżarzanie)

Klasa SA (Simulated Annealing) implementuje algorytm optymalizacji heurystycznej, który wykorzystuje idee symulowanego wyżarzania.

- **SA (Simulated Annealing):**
    - Klasa dziedzicząca po klasie abstrakcyjnej `Algorithm`.
    - Implementuje algorytm Symulowanego Wyżarzania (Simulated Annealing).
- **run:**
    - Metoda `run` przeprowadza optymalizację przez zadaną liczbę rund (`n_rounds`).
    - Inicjalizuje punkt pracy(wskazany przez `_work_point`) jako losowy punkt w przestrzeni parametrów modelu XGBoost.
    - W każdej rundzie generuje sąsiada punktu pracy (`new_point`) i oblicza jakość tego nowego punktu.
    - Jeśli nowy punkt ma lepszą jakość, staje się nowym punktem pracy.
    - Jeśli jakość nowego punktu jest gorsza, istnieje szansa na akceptację go jako nowego punktu pracy w zależności od różnicy jakości oraz aktualnej temperatury.
    - Aktualizuje najlepszy punkt, jeśli nowy punkt ma lepszą jakość.
    - Aktualizuje temperaturę w procesie optymalizacji zgodnie z ustalonym algorytmem chłodzenia.
    - Opcjonalnie loguje postęp optymalizacji na konsolę.
    - Zwraca krotkę zawierającą najlepszy punkt, jego jakość oraz historię jakości w kolejnych rundach.
- **init_params:**
    - Metoda inicjalizująca parametry algorytmu na podstawie przekazanego słownika `params`.
    - Wywołuje metodę `init_params` z klasy nadrzędnej do ustawienia podstawowych parametrów.

    - Ustawia dodatkowe parametry specyficzne dla algorytmu Symulowanego Wyżarzania, takie jak temperatura, algorytm chłodzenia i liczba rund z stałą temperaturą.

EA.py (Algorytm ewolucyjny)

Klasa `EA` implementuje algorytm Ewolucyjny (Evolutionary Algorithm), który jest heurystyczną metodą optymalizacji inspirowaną zasadami ewolucji biologicznej. Oto krótki opis klasy:

- **EA (Evolutionary Algorithm):**
    - Klasa dziedzicząca po klasie abstrakcyjnej `Algorithm`.
    - Implementuje algorytm Ewolucyjny.
- **run:**
    - Metoda `run` przeprowadza optymalizację przez zadaną liczbę rund (`n_rounds`).
    - Inicjalizuje populację punktów (`_population`) jako losowe punkty w przestrzeni parametrów modelu XGBoost.
    - W każdej rundzie przeprowadza proces reprodukcji, mutacji i sukcesji.
    - Reprodukcja polega na wyborze dwóch punktów z populacji, z których jeden jest wybierany jako "zwycięzca" na podstawie jakości, a następnie tworzony jest nowy punkt będący kopią zwycięzcy.
    - Mutacja polega na mutowaniu każdego punktu w populacji.
    - Sukcesja polega na wyborze najlepszych punktów spośród populacji i nowo utworzonych punktów mutowanych.
    - W każdej rundzie aktualizuje najlepszy punkt (`_best_point`) i jego jakość (`_best_quality`).
    - Opcjonalnie loguje postęp optymalizacji na konsolę.
    - Zwraca krotkę zawierającą najlepszy punkt, jego jakość oraz historię jakości w kolejnych rundach.
- **init_params:**
    - Metoda inicjalizująca parametry algorytmu na podstawie przekazanego słownika `params`.
    - Wywołuje metodę `init_params` z klasy nadrzędnej do ustawienia podstawowych parametrów.
    - Ustawia dodatkowe parametry specyficzne dla algorytmu Ewolucyjnego, takie jak prawdopodobieństwo mutacji, wskaźnik mutacji, rozmiar elity oraz rozmiar populacji.
- **`_init_population`:**
    - Inicjalizuje populację punktów jako losowe punkty w przestrzeni parametrów modelu XGBoost.
- **`_calculate_group_quality`:**
    - Oblicza jakość dla każdego punktu w populacji, korzystając z metody `_calculate_quality` z klasy nadrzędnej (`Algorithm`).
- **`_find_best`:**
    - Znajduje najlepszy punkt w populacji na podstawie jakości.

- **`_reproduction`:**
  - Przeprowadza proces reprodukcji, czyli tworzenia nowych punktów na podstawie zwycięzców pojedynków.
  - Każdy nowy punkt jest kopią zwycięzcy pojedynku.
- **`_mutation`:**
  - Przeprowadza mutację dla każdego punktu w populacji.
  - Prawdopodobieństwo mutacji jest kontrolowane przez parametr `_mutation_prob`.
  - Współczynnik mutacji, `_mutation_rate`, wpływa na stopień modyfikacji parametrów punktów podczas mutacji.
- **`_succession`:**
  - Przeprowadza proces sukcesji, czyli wybierania najlepszych punktów z obecnej populacji i nowo utworzonych punktów mutowanych.
  - Zachowuje elity z obu populacji.
- **`_sort`:**
  - Metoda sortująca populację i jakość punktów w populacji w porządku malejącym na podstawie jakości.

Eksperymenty

Zdecydowaliśmy, że eksperymenty podzielimy na dwie fazy:

- wstępna - badamy ogólne zachowanie algorytmów optymalizacji, próbujemy wyciągnąć wnioski na tej podstawie np. które parametry są ważne, z których można zrezygnować, jakie kombinacje parametrów mają sens.
- szczegółowa - wybieramy kilka potencjalnie najlepszych rozwiązań i skupiamy się na osiągnięciu jak najlepszych rezultatów np. poprzez dostrojenie hiperparametrów algorytmu optymalizacji, wybór innych hiperparametrów XGBoost itp.

Faza wstępna

Po wstępnym zaimplementowaniu algorytmów sprawdziliśmy ich działanie dla powyższych cech:

```
N_RUNS = 10
ALGORITHMS = RS, SA, EA
METRICS = 'gini', 'accuracy'
DATASETS = 'smoker_status', 'porto_seguro'

EA_EPOCHS = 50
RS_SA_EPOCHS = 300
```

Każdą kombinację parametrów (Algorytm, Metryka, Zbiór danych) uruchamiamy 10 razy, aby uwzględnić czynnik losowy. Algorytm ewolucyjny uruchamiamy na 50 rund z uwagi na jego złożoność, a pozostałe, prostsze na 300 rund.

Hiperparametry algorytmów

Wstępne hiperparametry algorytmów optymalizacji zostały wybrane arbitralnie. Stałe parametry dla każdego modelu XGBoost były następujące:

```
    'objective': 'binary:logistic',
    'eval_metric': 'logloss',
    'device': 'cuda'
```

oraz

```
NUM_BOOST_ROUND = 10
```

Dzięki zdefiniowaniu tych stałych, każdy algorytm optymalizacji mógł korzystać z takiego samego modelu XGBoost.

Przeszukiwanie losowe (RS)

Brak hiperparametrów.

Symulowane wyżarzanie (SA)

```
    "temperature": 100,
    "cooling": 0.95,
    "rounds_with_const_temperature": 10,
```

Algorytm ewolucyjny (EA)

```
    "mutation_prob": 0.9,
    "mutation_rate": 0.1,
    "population_size": 10,
    "elite_size": 1,
```

Parametry te będą podlegać dostrajaniu w ciągu dalszych eksperymentów.

Obserwowane zmienne

Po każdym uruchomieniu algorytmu zostaną zapisane:

- najlepszy punkt
- najlepsza jakość podczas optymalizacji
- jakość najlepszego punktu na zbiorze testowym
- przebieg optymalizacji - najlepszy punkt w każdej epoce

Dane zostaną zapisane do pliku JSON, aby możliwa była ich analiza w oderwaniu od uruchomienia algorytmów optymalizacyjnych.

Przebieg eksperymentów

Największym problemem był czas, ponieważ obliczenie funkcji celu zajmowało ponad sekundę. Musieliśmy wielokrotnie dostosowywać parametry tak, aby znaleźć rozsądną granicę między długością optymalizacji oraz jakością.

Oszacowanie liczby wywołań funkcji celu

W jednej iteracji:

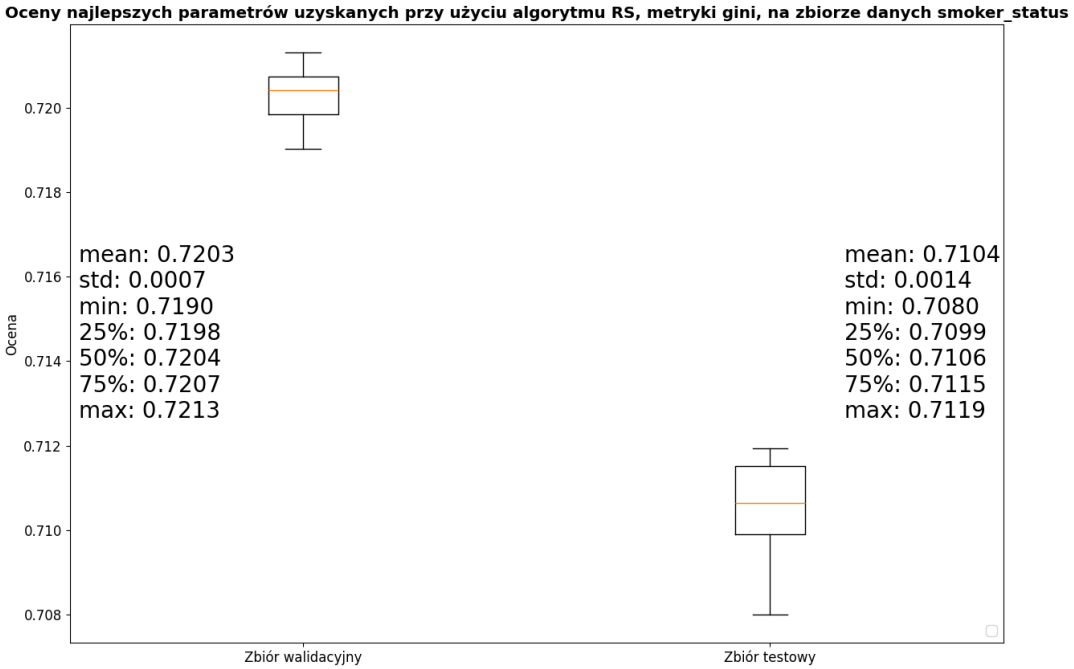
- Przeszukiwanie losowe - $300 \cdot 1 = 300$
- Symulowane wyżarzanie - $1 + 300 \cdot 1 = 301$
- Algorytm ewolucyjny - $10 + 30 \cdot 10 = 310$

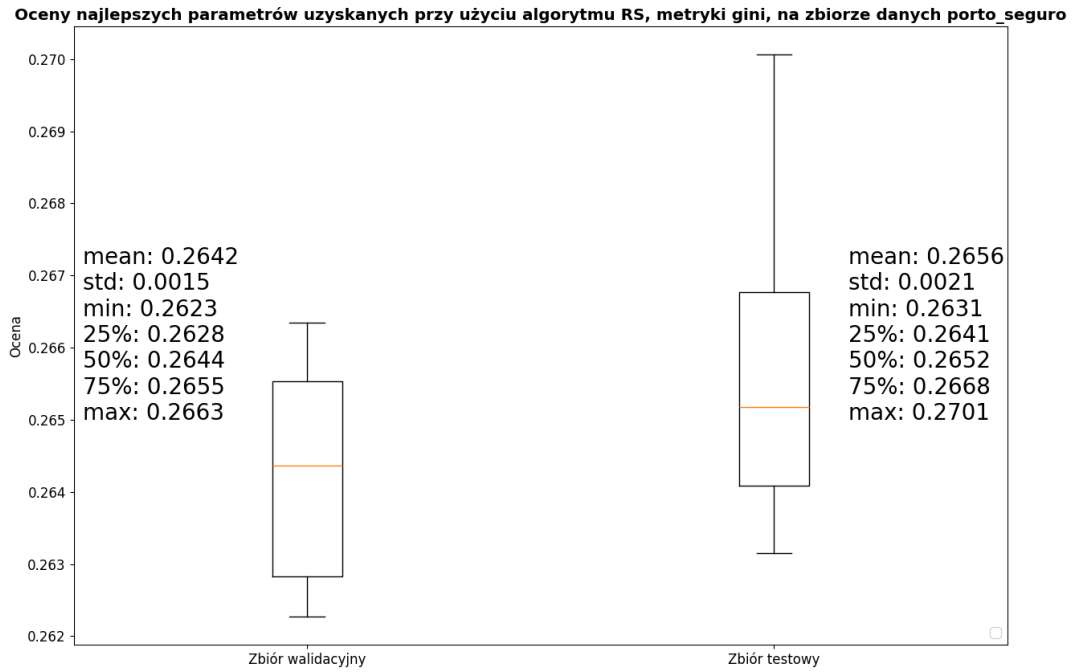
Każdy z algorytmów otrzymał podobną ilość wywołań funkcji celu, co pozwala porównać je względnie obiektywnie.
Wywołań łącznie: $10 \cdot (300 + 301 + 310) = 9110$

Obserwacje

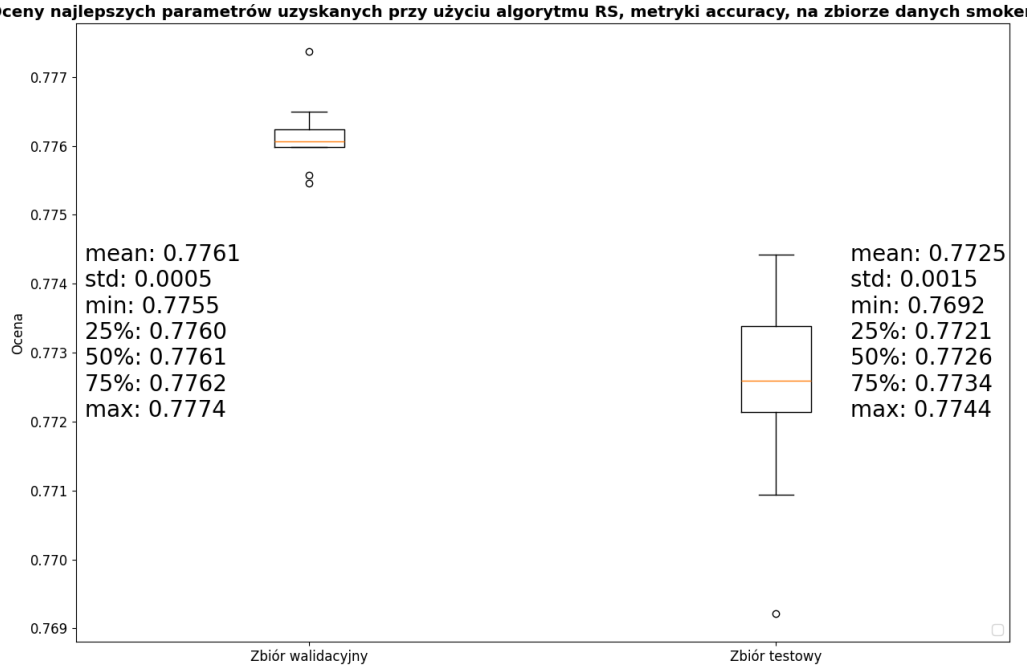
Znalezione najlepsze rezultaty

Przeszukiwanie losowe

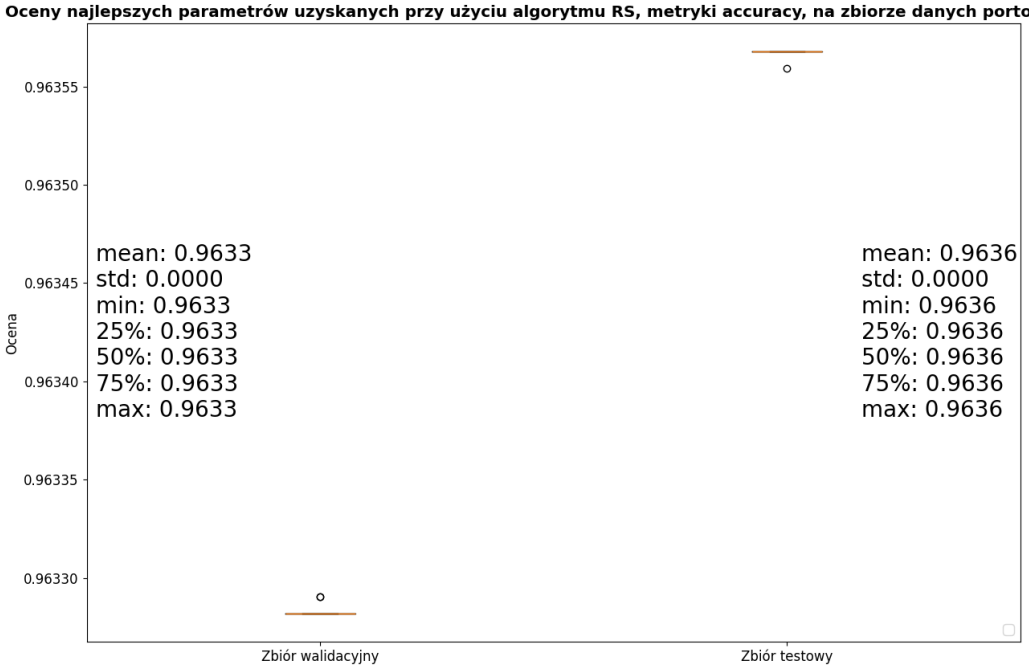




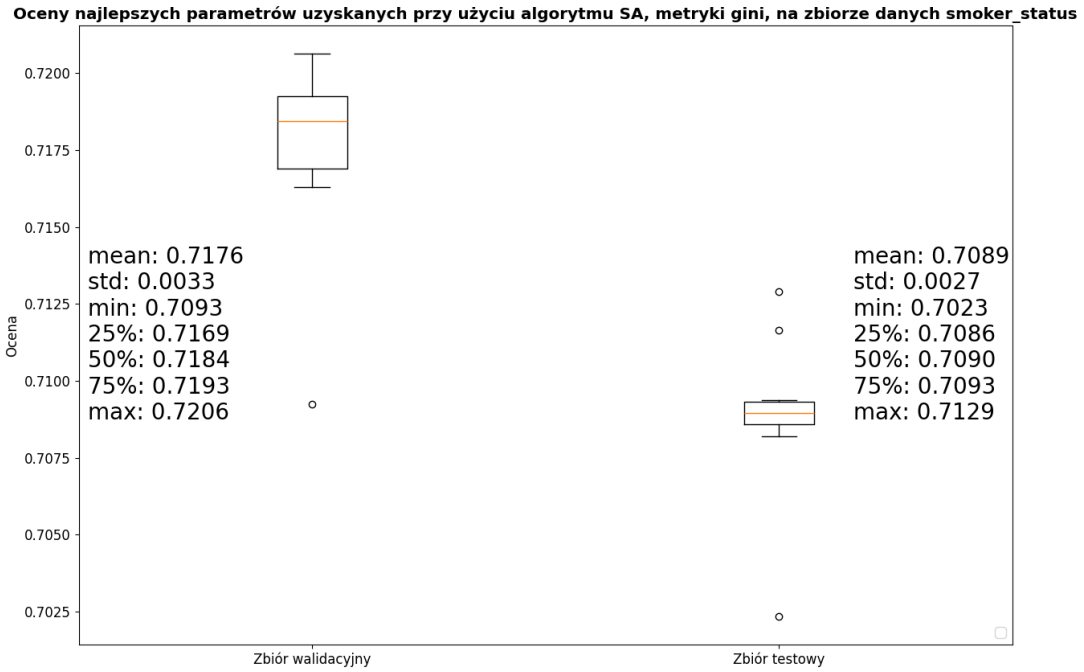
Oceny najlepszych parametrów uzyskanych przy użyciu algorytmu RS, metryki accuracy, na zbiorze danych smoker_status

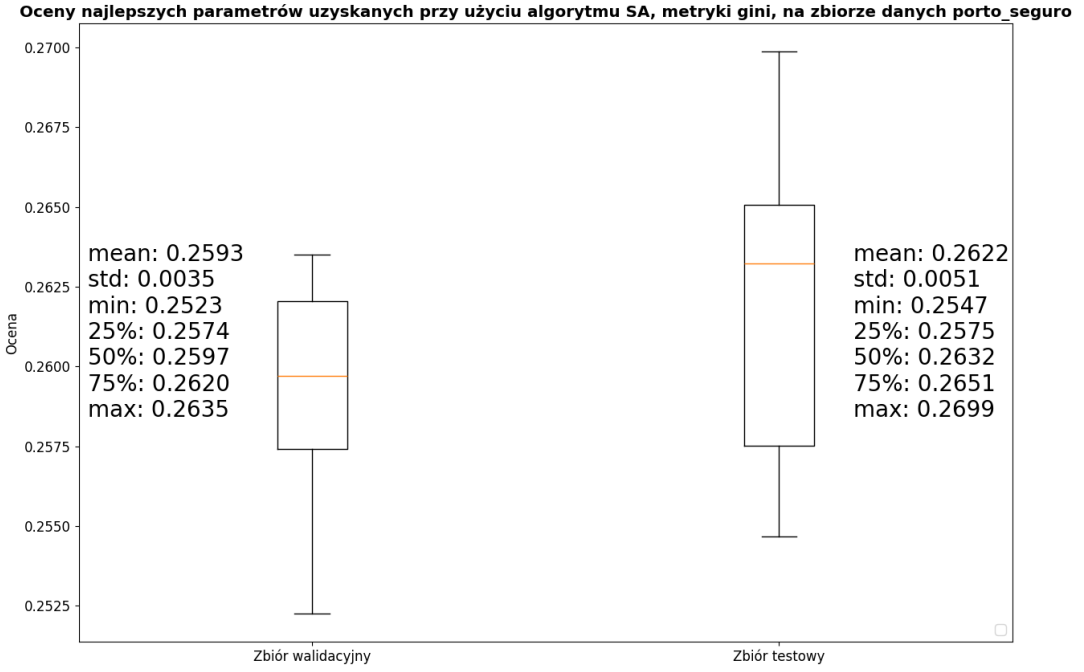


Oceny najlepszych parametrów uzyskanych przy użyciu algorytmu RS, metryki accuracy, na zbiorze danych porto_seguro

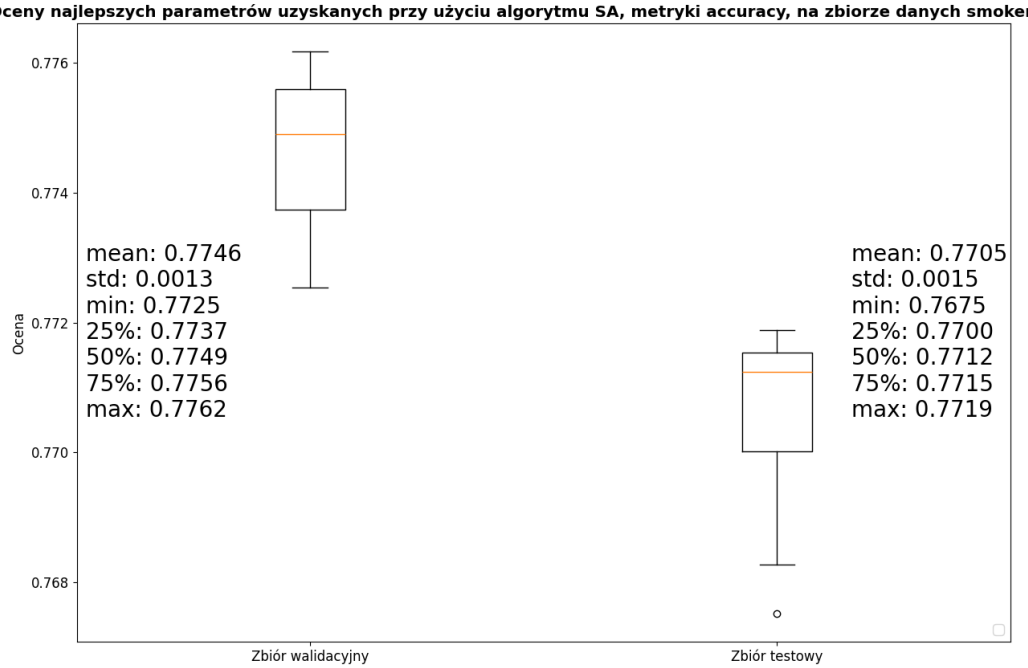


Symulowane wyżarzanie

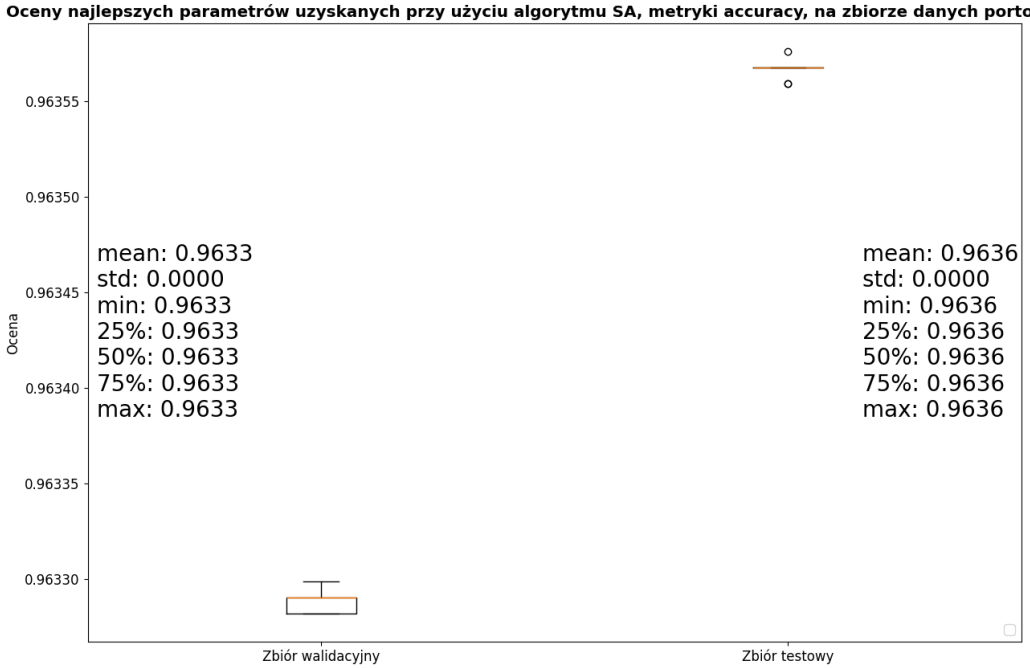




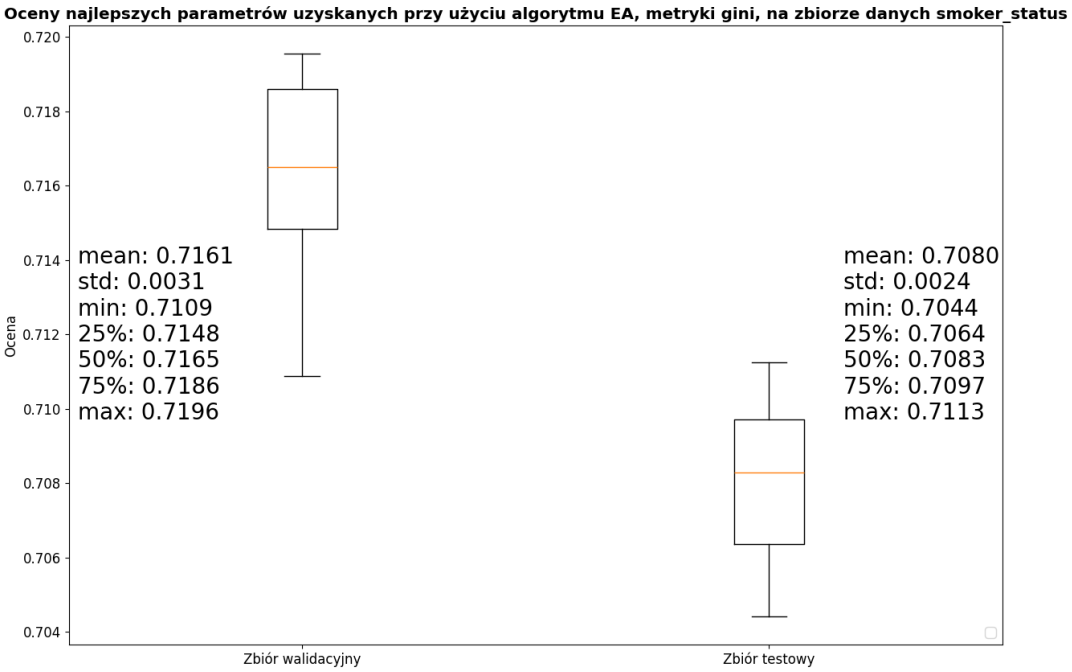
Oceny najlepszych parametrów uzyskanych przy użyciu algorytmu SA, metryki accuracy, na zbiorze danych smoker_status

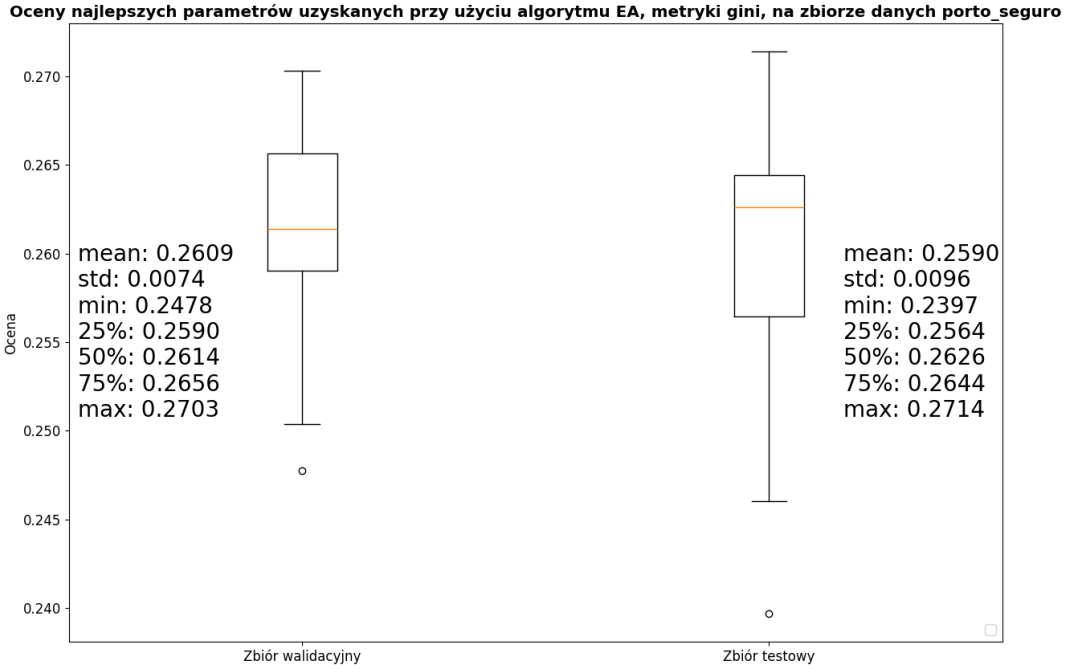


Oceny najlepszych parametrów uzyskanych przy użyciu algorytmu SA, metryki accuracy, na zbiorze danych porto_seguro

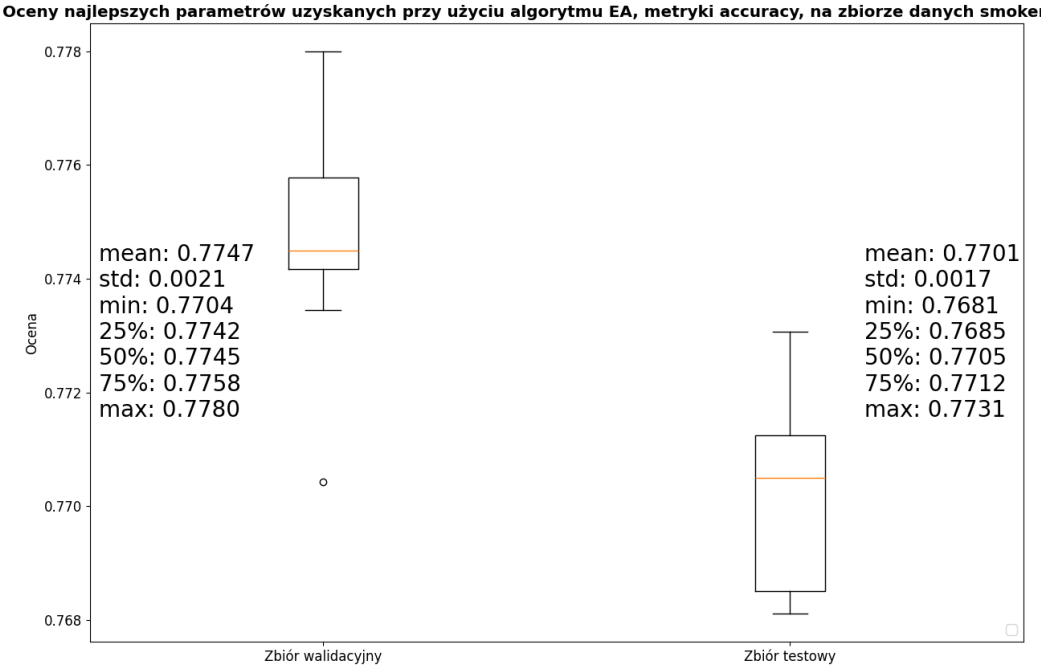


Algorytm ewolucyjny

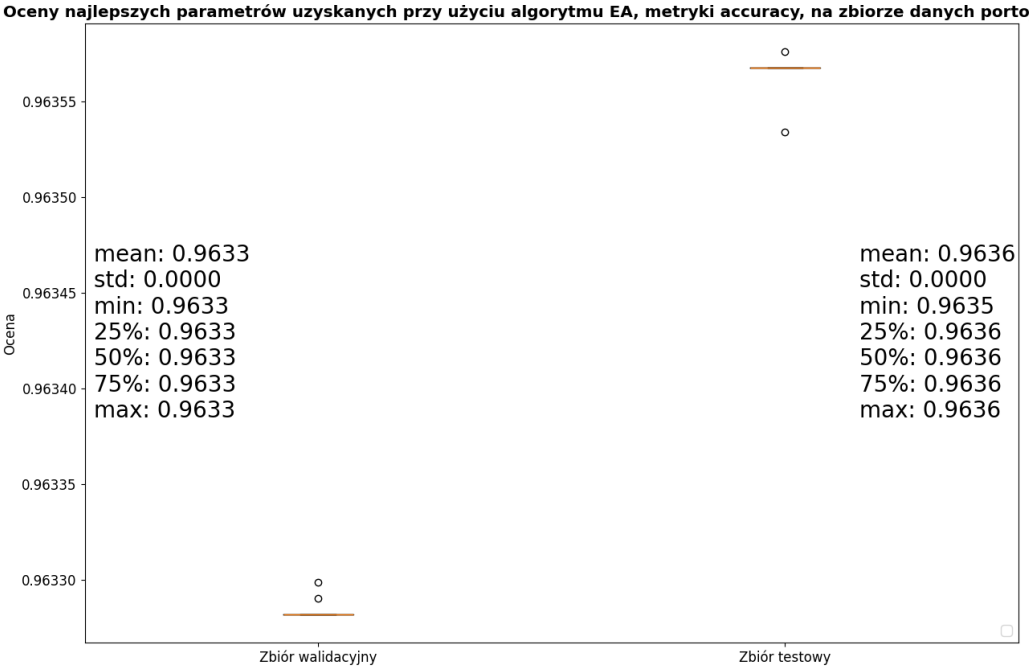




Oceny najlepszych parametrów uzyskanych przy użyciu algorytmu EA, metryki accuracy, na zbiorze danych smoker_status



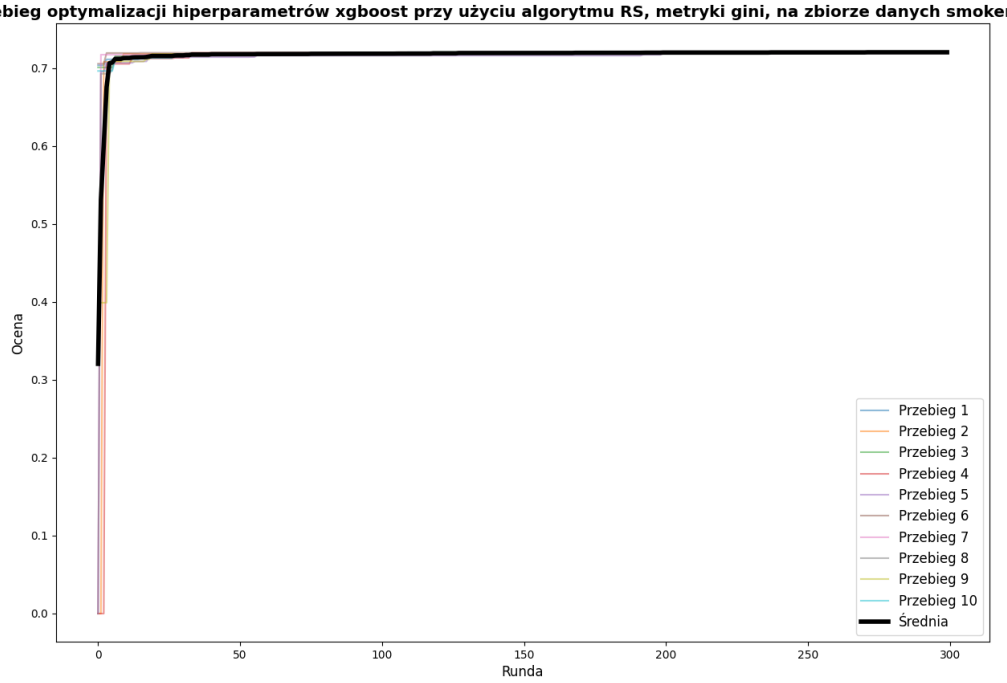
Oceny najlepszych parametrów uzyskanych przy użyciu algorytmu EA, metryki accuracy, na zbiorze danych porto_seguro



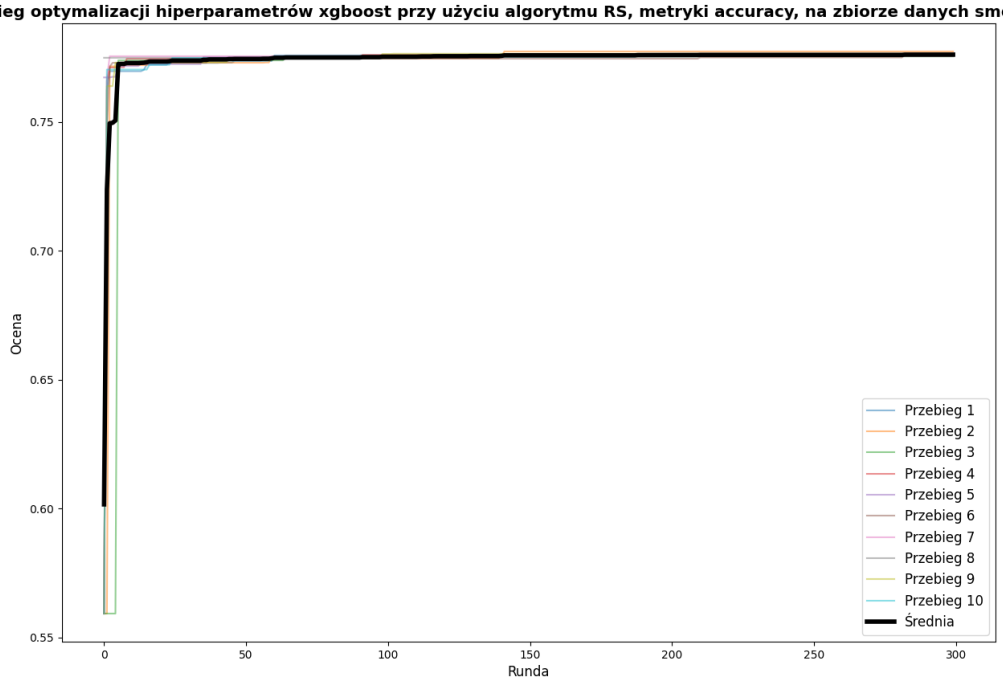
Przebieg optymalizacji

Przeszukiwanie losowe

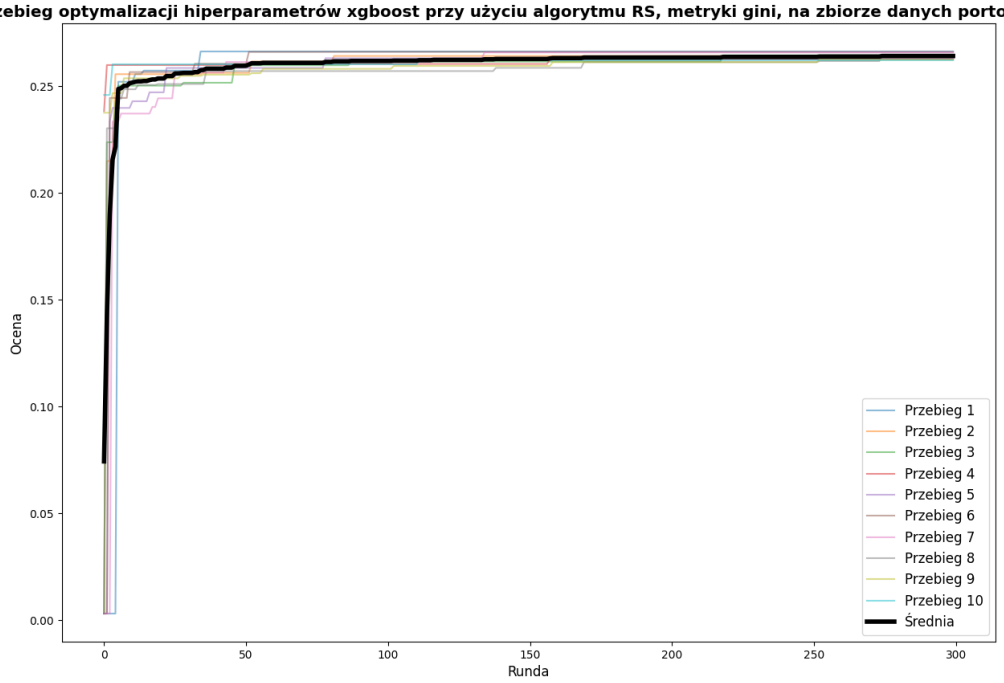
Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu RS, metryki gini, na zbiorze danych smoker_status.



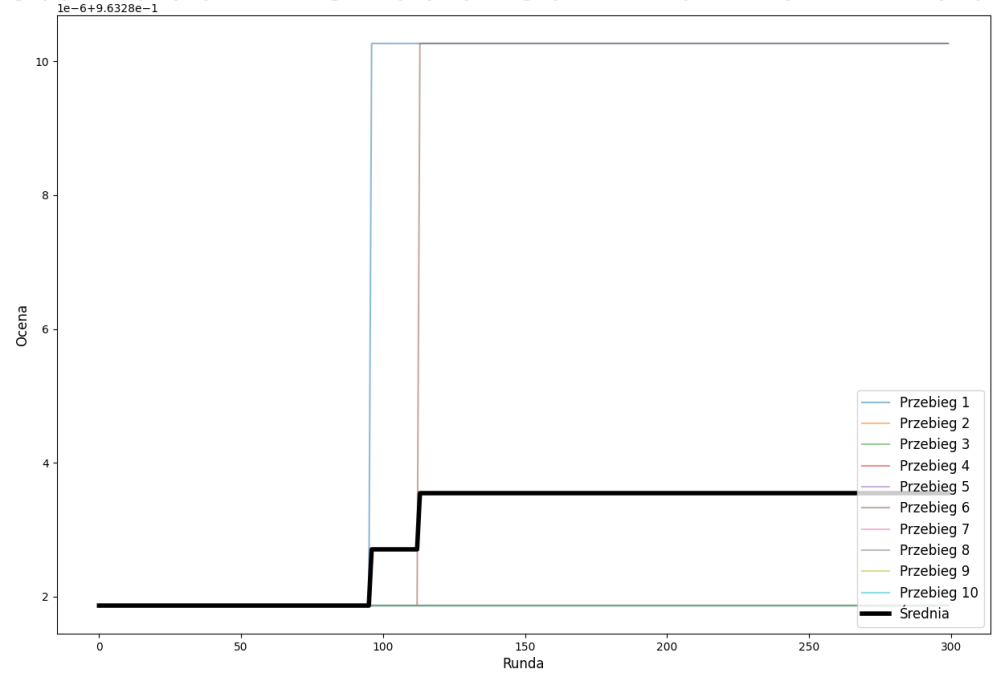
Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu RS, metryki accuracy, na zbiorze danych smoker_status.



Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu RS, metryki gini, na zbiorze danych porto_seguro.

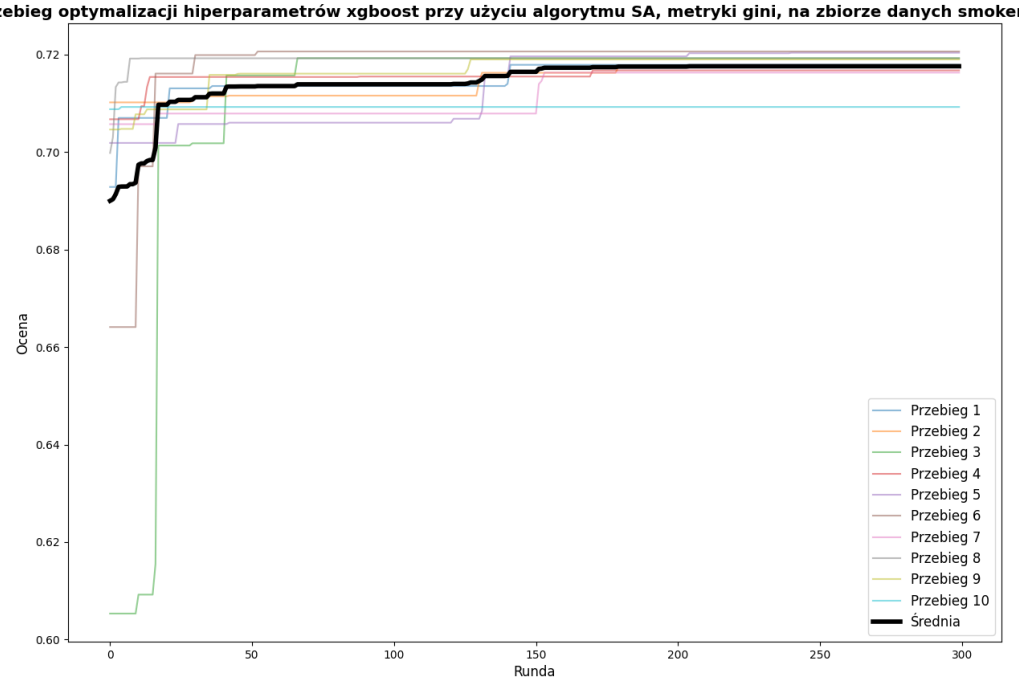


Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu RS, metryki accuracy, na zbiorze danych porto_seguro.

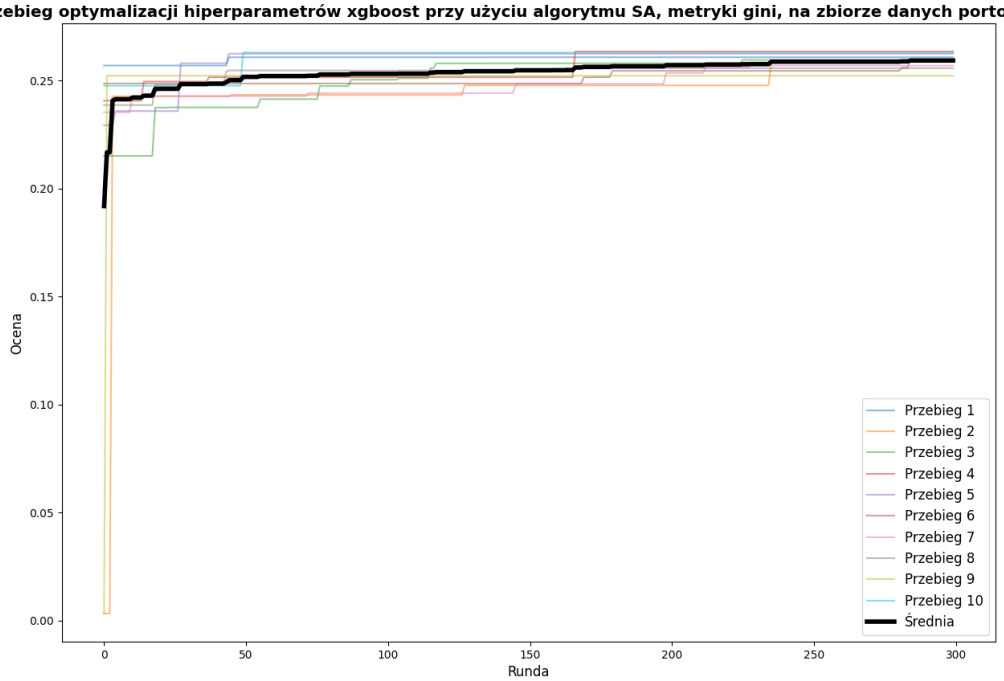


Symulowane wyżarzanie

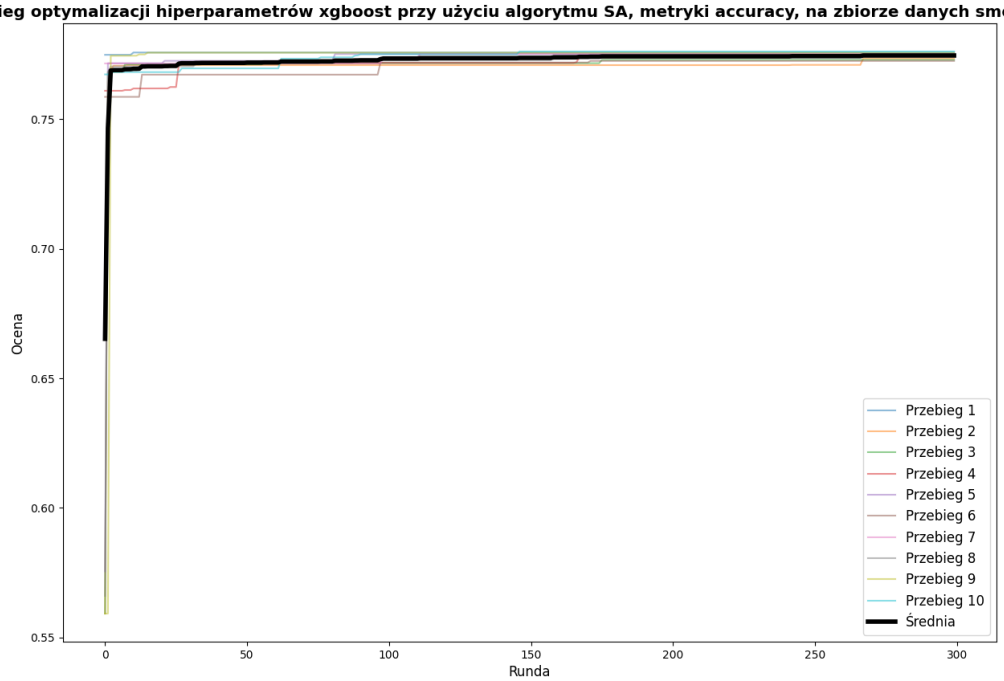
Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu SA, metryki gini, na zbiorze danych smoker_status.



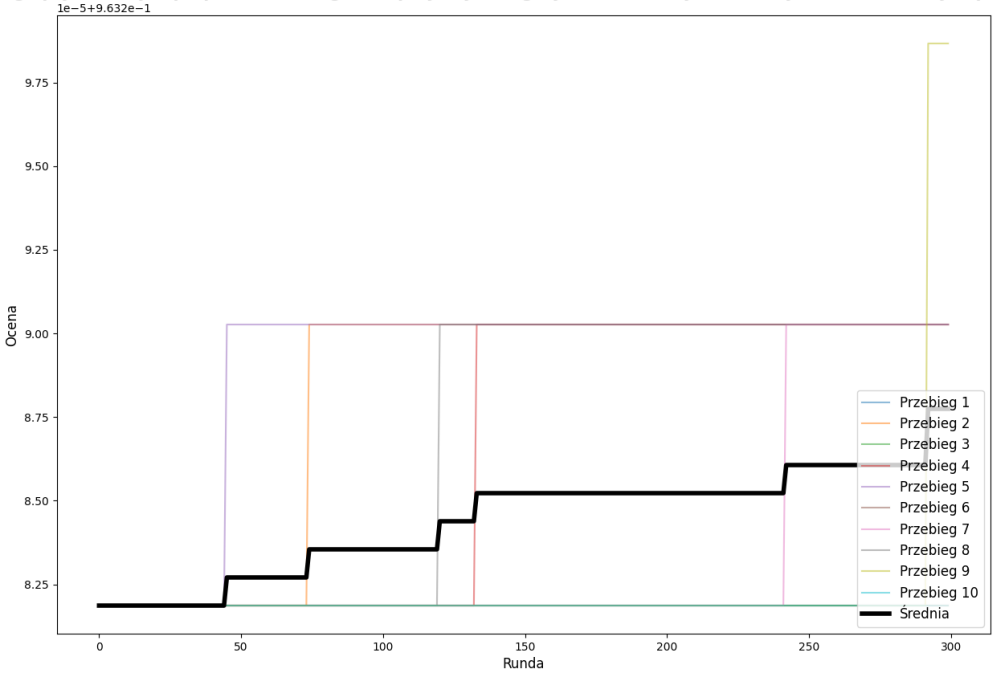
Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu SA, metryki gini, na zbiorze danych porto_seguro.



Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu SA, metryki accuracy, na zbiorze danych smoker_status.

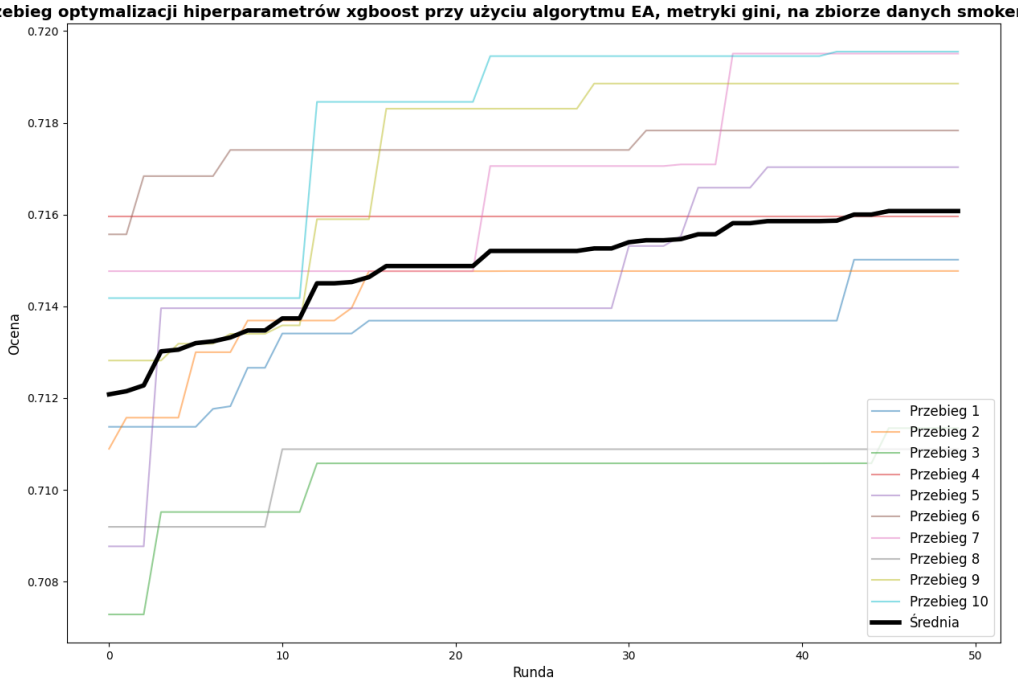


Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu SA, metryki accuracy, na zbiorze danych porto_seguro.

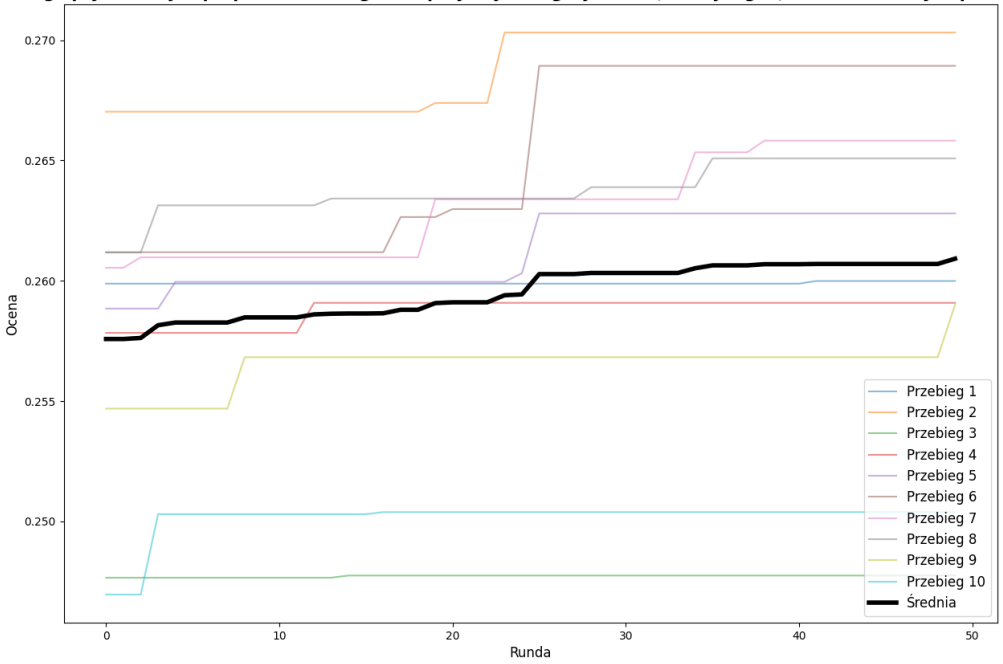


Algorytm ewolucyjny

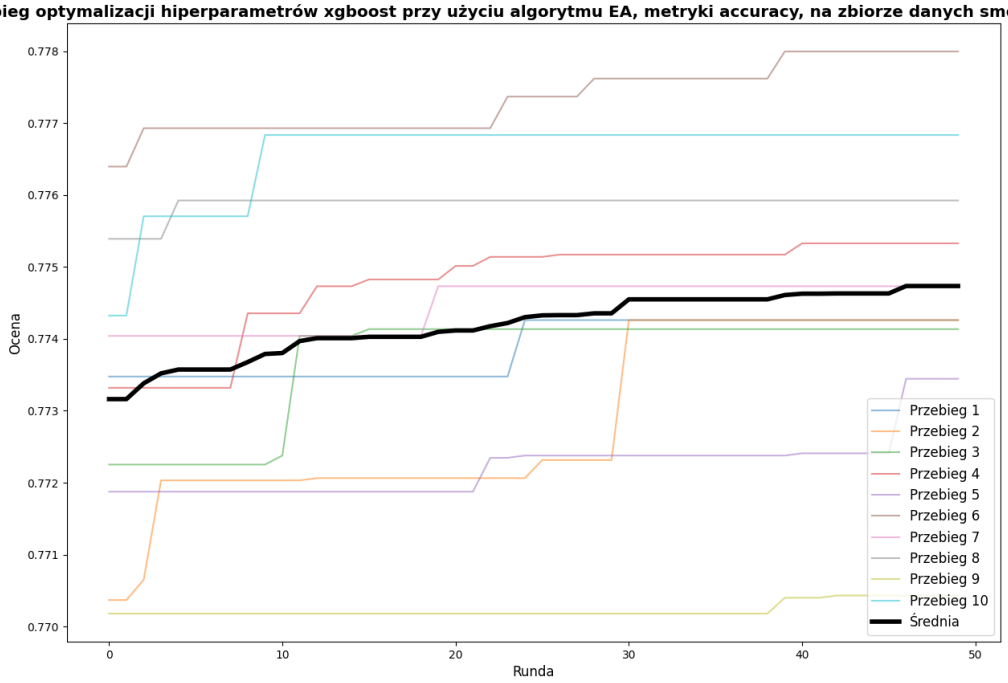
Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu EA, metryki gini, na zbiorze danych smoker_status.



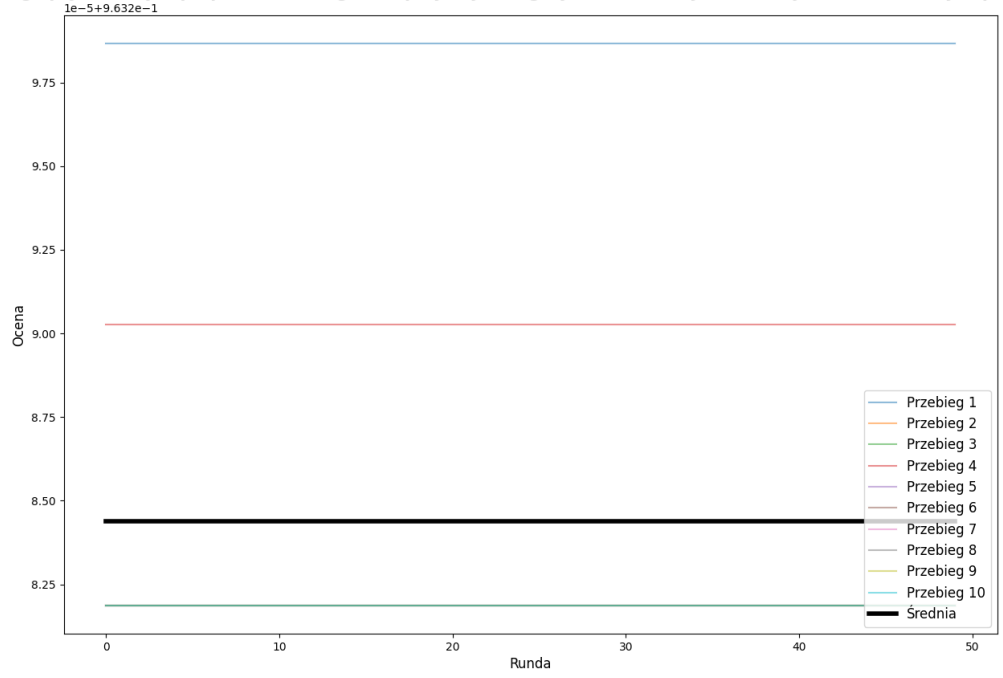
Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu EA, metryki gini, na zbiorze danych porto_seguro.



Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu EA, metryki accuracy, na zbiorze danych smoker_status.



Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu EA, metryki accuracy, na zbiorze danych porto_seguro.



Wnioski

1. Metryka accuracy nie ma sensu dla zbioru porto_seguro. Skuteczność zawsze jest wysoka, klasa większościowa stanowi 96.4%. Wyniki osiągane dla tej metryki są

zawsze bardzo podobne, zbliżone do przewidywania zawsze klasy większościowej. Optymalizacja nic nie daje. Warto odrzucić tą metrykę dla tego zbioru.

2. Osiągane wyniki nie są zbyt dobre. Dla domyślnych parametrów algorytmu XGBoost otrzymaliśmy następujące wyniki:

   porto_seguro:
   a. Accuracy: 0.96 (nie bierzemy pod uwagę)
   b. Gini score: 0.26

   smoker_status:
   a. Accuracy: 0.78
   b. Gini score: 0.72

   Algorytmy optymalizacji znalazły następujące rezultaty:

   porto_seguro:
   a. Accuracy: 0.96 (nie bierzemy pod uwagę)
   b. Gini score: 0.248 - 0.27

   smoker_status:
   a. Accuracy: 0.768 - 0.778
   b. Gini score: 0.709 - 0.721

   Znalezione zestawy parametrów w najlepszym wypadku dają zbliżone wyniki do parametrów domyślnych algorytmu XGBoost. Prawdopodobnie są tego dwie przyczyny:
   a. Domyślne parametry XGBoost są dobrane tak, aby dało się korzystając z nich zbudować zadowalający model.
   b. Przestrzeń poszukiwań jest zbyt duża. Nawet wstępne ograniczenie ilości parametrów nie miało znaczącego, pozytywnego wpływu na jakość optymalizacji. Można też zauważyć, że im mniej parametrów rozpatrujemy tym bardziej zbliżamy się do modelu domyślnego, więc dalsze zmniejszanie przestrzeni poszukiwań nie ma sensu.

   Jeżeli domyślny model jest już dobry, to być może dobrym podejściem byłoby zainicjowanie algorytmów optymalizacyjnych domyślnymi parametrami modelu lub parametrami zbliżonymi do nich. Wtedy moglibyśmy dobrać hiperparametry tych algorytmów tak, aby szukały punktów w otoczeniu wartości domyślnych. W tym przypadku rezygnujemy z algorytmu przeszukiwania losowego, ponieważ nie jest on inicjowany żadnym punktem.

3. Na niektórych wykresach przebiegu optymalizacji można zauważyć znaczący skok jakości (z ok. 0.6 na ok. 0.7). Występuje to w algorytmach RS i SA. Po przeanalizowaniu logów programu doszliśmy do wniosku, że odpowiada za to zmiana parametru ‘booster’ z ‘gblinear’ na ‘gbtree’. Warto też zauważyć, że wśród wyników optymalizacji czyli najlepszych znalezionych punktów w danym uruchomieniu, parametr ‘booster’ przyjmował wartość ‘gblinear’ tylko gdy algorytm był uruchamiany z metryką ‘accuracy’ na zbiorze porto_seguro, co jak wykazaliśmy wcześniej, dawało losowe wyniki i trudno nadać im większe znaczenie.

   Średnio 50% punktów korzystało z liniowego ‘boostera’. Rezygnując z niego jesteśmy w stanie zaoszczędzić dużą ilość zasobów, które możemy przeznaczyć np.

na dłuższe działanie algorytmu ewolucyjnego lub np zwiększenie jego populacji. Dzięki temu w tym samym czasie będzie mógł przeszukać przestrzeń dokładniej.

Podsumowanie

W etapie eksperymentów szczegółowych postaramy się spełnić następujące założenia:

1. Rezygnujemy z metryki ‘accuracy’ dla zbioru porto_seguro.
2. Rezygnujemy z ‘booster’: ‘gblinear’
3. Rezygnujemy z algorytmu przeszukiwania losowego
4. Inicjujemy algorytmy wartościami z otoczenia wartości domyślnej.

Faza szczegółowa

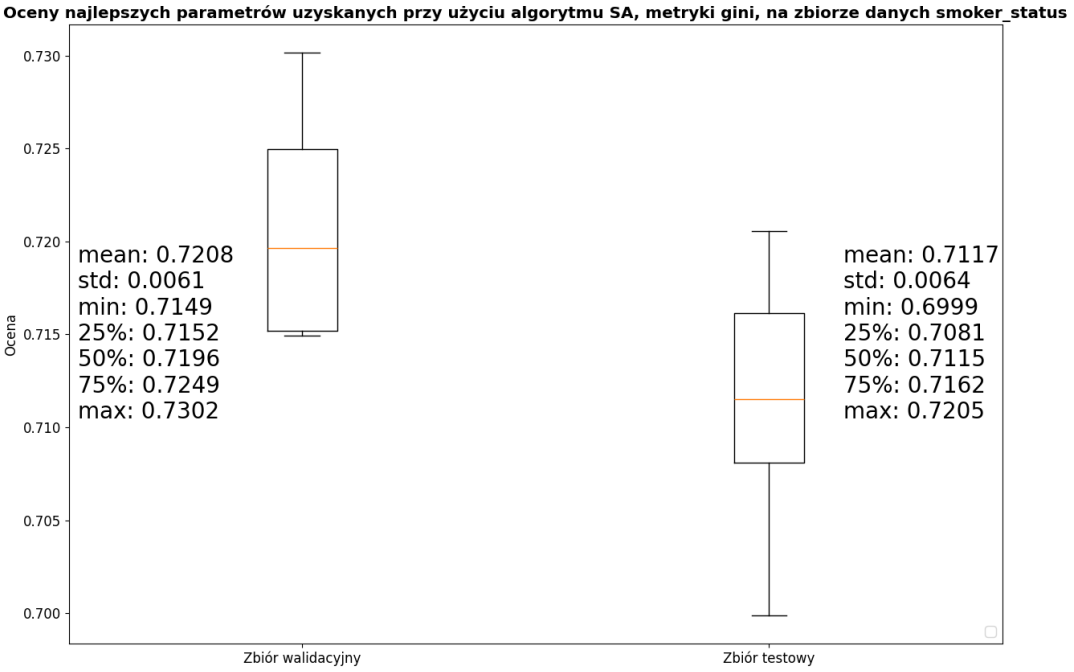
Skupiliśmy się na algorytmie ewolucyjnym oraz symulowanym wyżarzaniu. Z dokumentacji biblioteki xgboost pobraliśmy domyślne parametry algorytmu. Zmodyfikowaliśmy nasz program tak, aby każdy algorytm przeszukiwania był inicjowany tymi parametrami. Zrezygnowaliśmy z boostera ‘gblinear’, ponieważ nie dawał żadnych wymiernych korzyści, a zajmował połowę zasobów czasowych

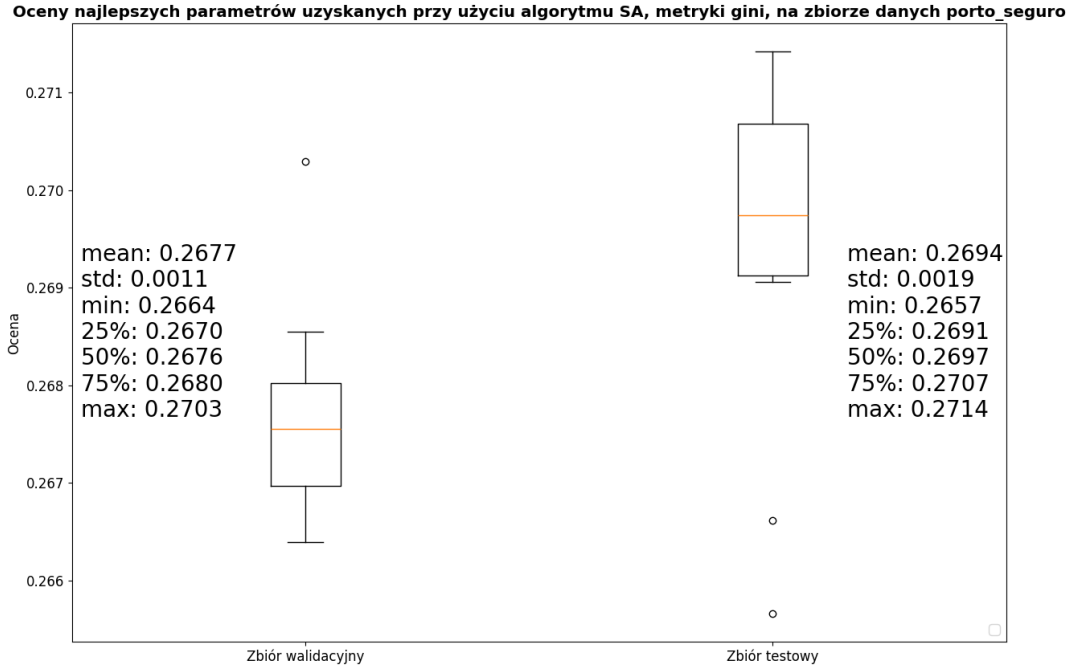
Hiperparametry algorytmów

Obserwacje

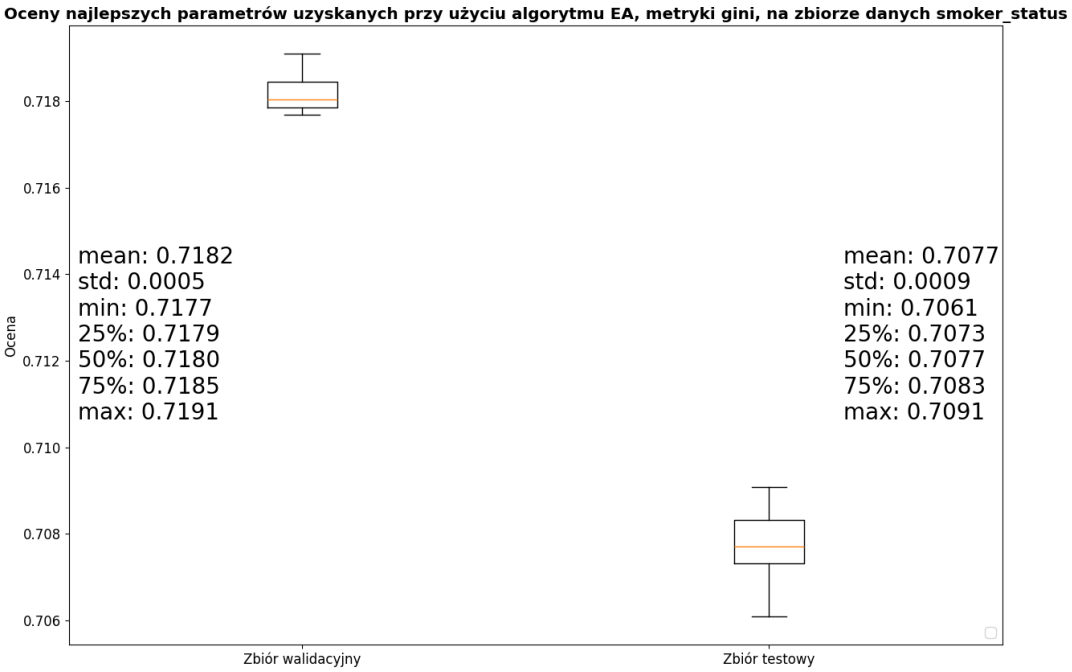
Znalezione najlepsze rezultaty

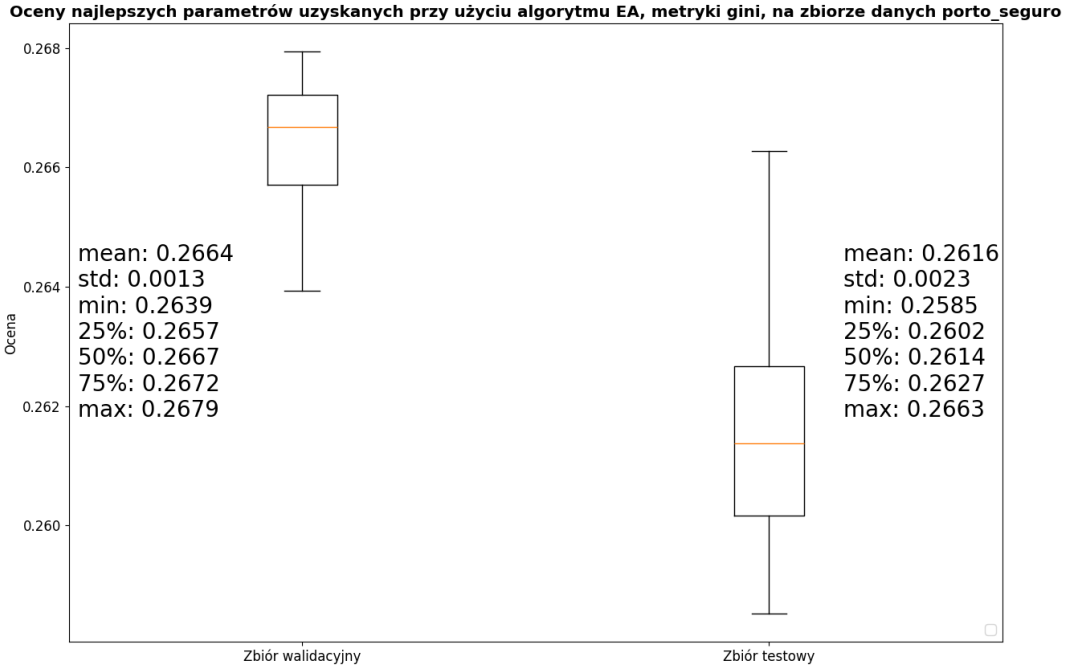
Symulowane wyżarzanie





Algorytm ewolucyjny

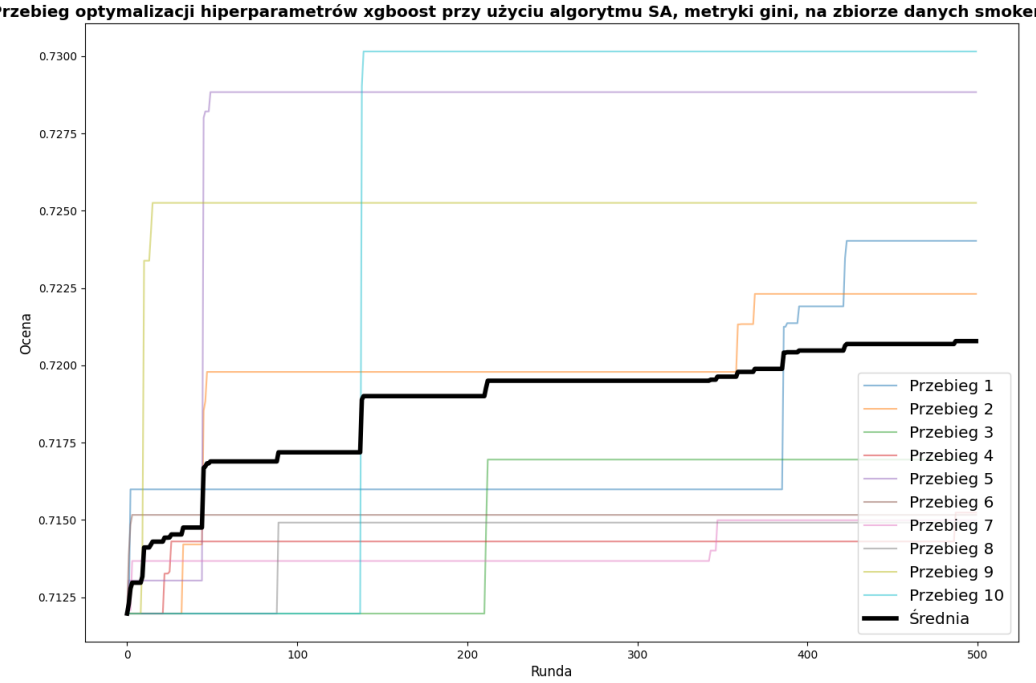




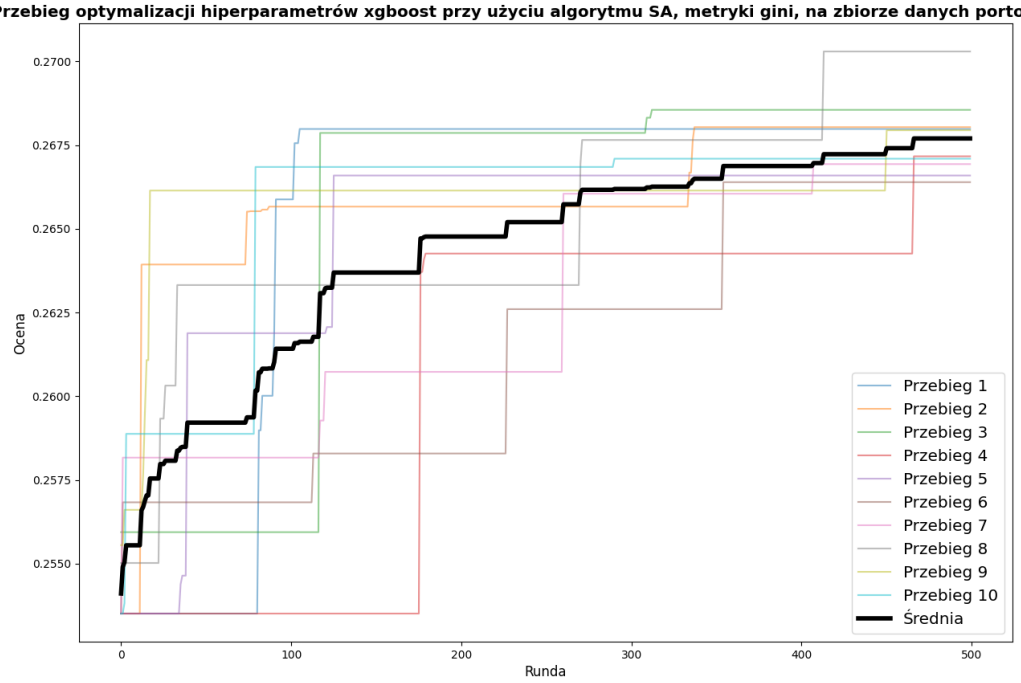
Przebieg optymalizacji

Symulowane wyżarzanie

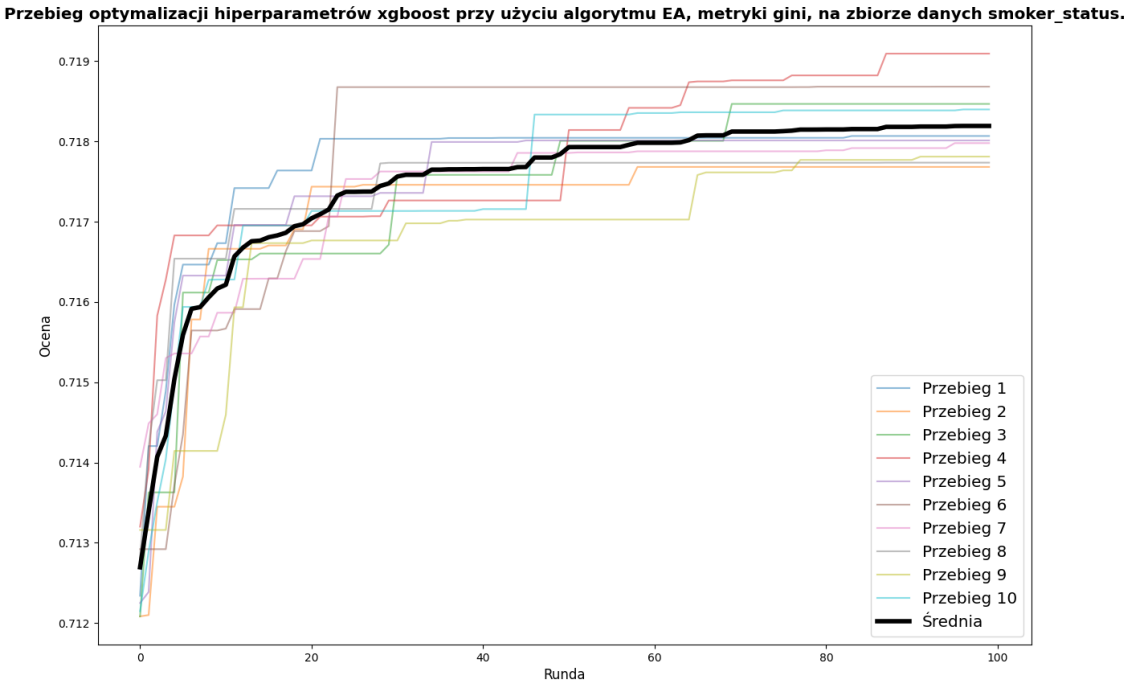
Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu SA, metryki gini, na zbiorze danych smoker_status.

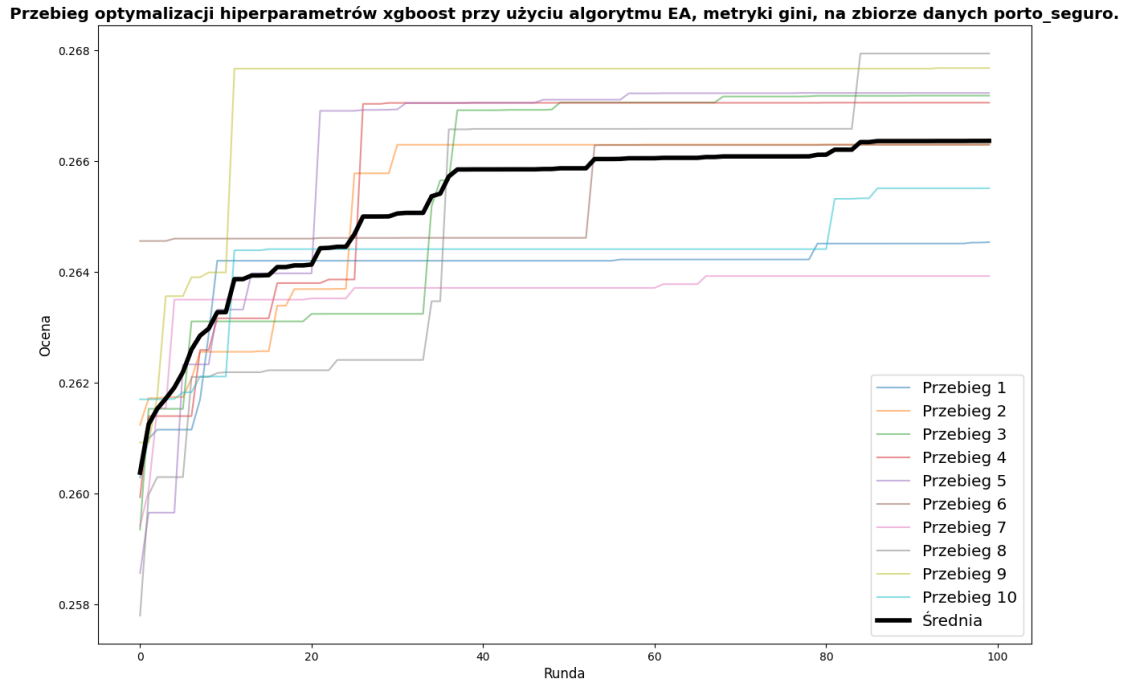


Przebieg optymalizacji hiperparametrów xgboost przy użyciu algorytmu SA, metryki gini, na zbiorze danych porto_seguro.



Algorytm ewolucyjny





Najlepsze parametry algorytmu XGBoost

Poniżej przedstawiamy wartości parametrów, dla których algorytm XGBoost osiągnął najlepszą jakość klasyfikacji na zbiorze testowym.

Metryka Gini, Zbiór Smoker Status

Jakość na zbiorze testowym: 0.7205
Algorytm: SA
Parametry:

"booster": "gbtree",
"learning_rate": 0.44672685772902465,
"min_split_loss": 3.151639067699963,
"max_depth": 20,
"min_child_weight": 4.270318315887933,
"max_delta_step": 3.8318224172763227,
"subsample": 0.8150585506365398,
"reg_lambda": 1.3086560582963391,
"reg_alpha": 5.491929660584046,
"refresh_leaf": 0,
"grow_policy": "lossguide",
"max_leaves": 0,
"max_bin": 277,
"num_parallel_tree": 8

Metryka Gini, Zbiór Porto Seguro

Jakość na zbiorze testowym: 0.2714
Algorytm: SA
Parametry:

"booster": "gbtree",
"learning_rate": 0.5394889061658026,
"min_split_loss": 11.45391288874038,
"max_depth": 15,
"min_child_weight": 8.451923662989186,
"max_delta_step": 5.292893287078815,
"subsample": 0.7009570493769901,
"reg_lambda": 1.3098532210002334,
"reg_alpha": 0.21972098319276268,
"refresh_leaf": 1,
"grow_policy": "lossguide",
"max_leaves": 0,
"max_bin": 290,
"num_parallel_tree": 10

Podsumowanie

W obu przypadkach najlepszy punkt został znaleziony przez algorytm symulowanego wyżarzania. Widzimy, że najlepsze parametry w przypadku obu zbiorów danych są podobne. W szczególności atrybuty: learning_rate, reg_lambda, grow_policy, max_leaves, max_bin, num_paralel_tree.

Wnioski

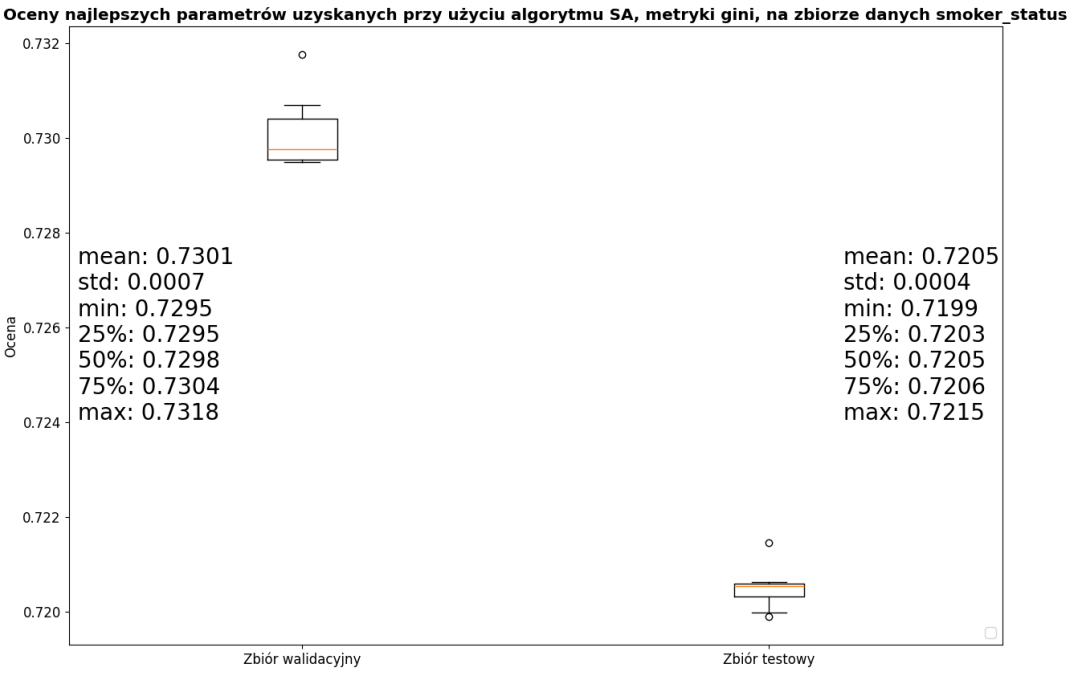
Osiągane rezultaty nieznacznie się poprawiły. Zaobserwowaliśmy, że wyniki są bardziej zbliżone do siebie, mniejsze jest odchylenie standardowe. Przyczyną tego prawdopodobnie jest wybór punktu startowego. Jeżeli populacja na samym początku jest skupiona w jednym punkcie, to wyniki algorytmu optymalizacji będą bardziej zbliżone do siebie, niż przy rozpoczynaniu z całkowicie losową populacją.

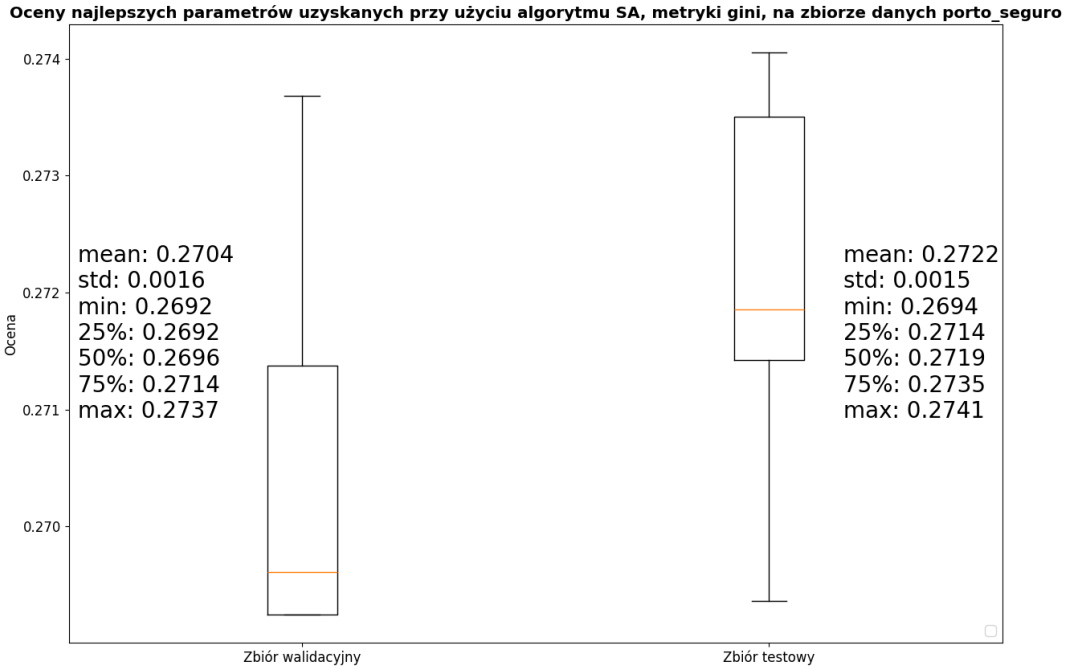
Faza końcowa

Zainicjujemy algorytm symulowanego wyżarzania najlepszymi dotychczas znalezionymi punktami i uruchomimy go przez większą ilość rund.

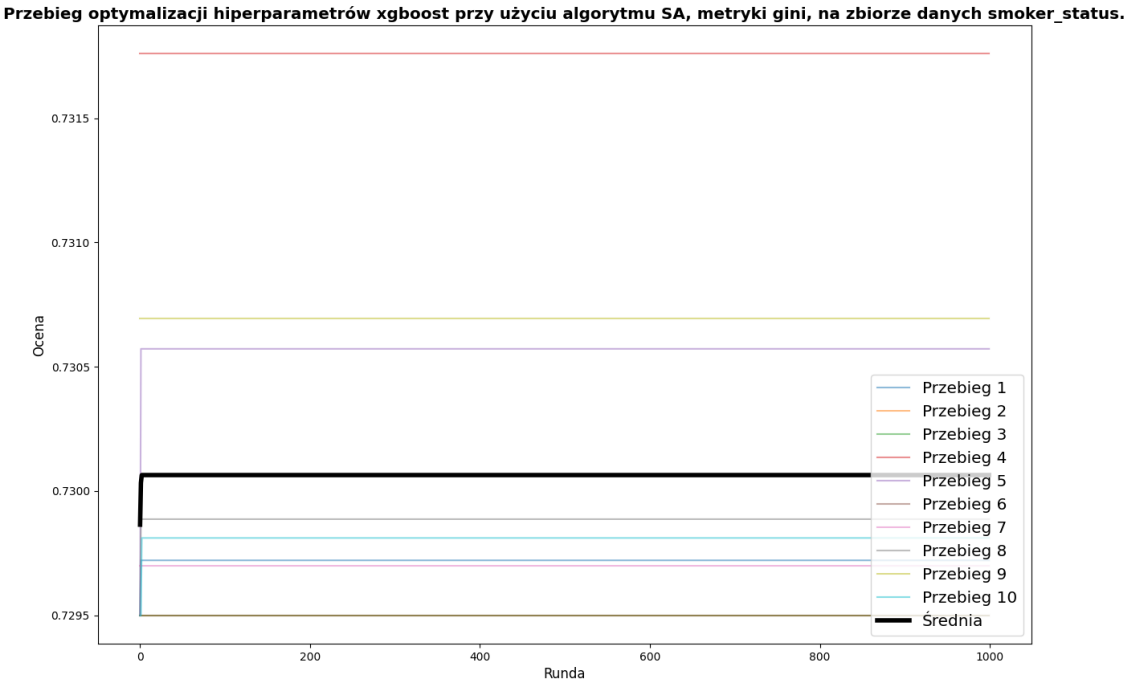
Obserwacje

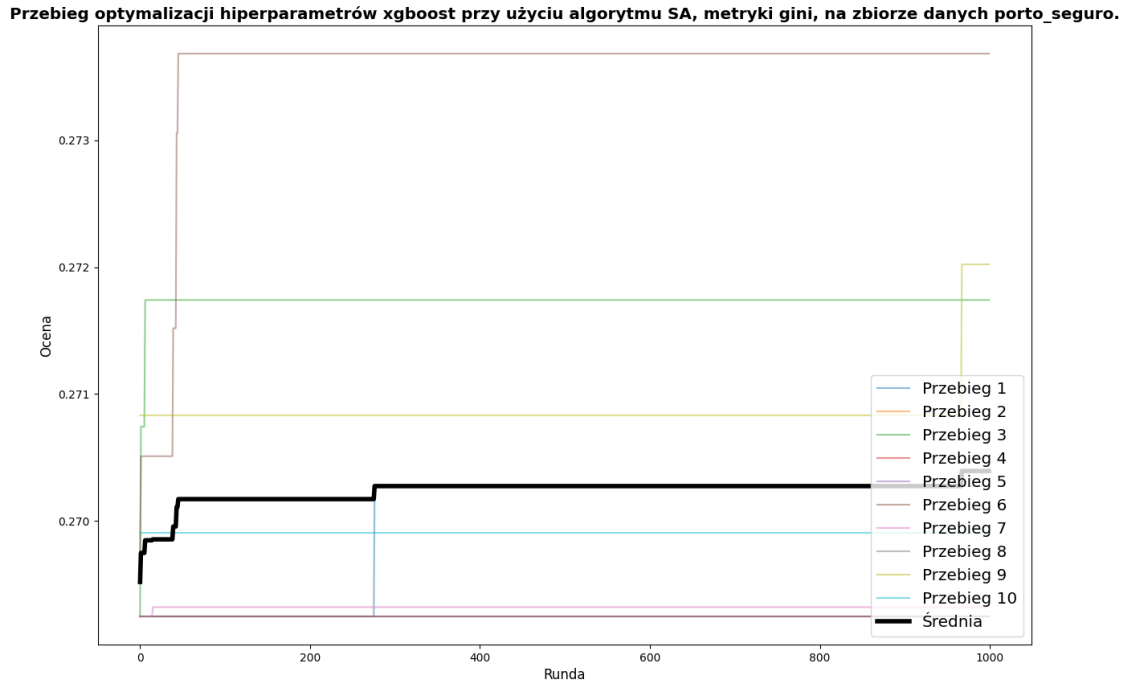
Znalezione najlepsze rezultaty





Przebieg optymalizacji





Najlepsze parametry algorytmu XGBoost

Metryka Gini, Zbiór Smoker Status

Jakość na zbiorze testowym: 0.2741
Parametry:

"booster": "gbtree",
"learning_rate": 0.7266841497518138,
"min_split_loss": 3.151639067699963,
"max_depth": 20,
"min_child_weight": 4.270318315887933,
"max_delta_step": 3.8318224172763227,
"subsample": 0.8150585506365398,
"reg_lambda": 1.3086560582963391,
"reg_alpha": 5.491929660584046,
"refresh_leaf": 0,
"grow_policy": "lossguide",
"max_leaves": 0,
"max_bin": 277,
"num_parallel_tree": 8

Metryka Gini, Zbiór Porto Seguro

Jakość na zbiorze testowym: 0.7215
Parametry:

"booster": "gbtree",
"learning_rate": 0.49262014939478915,
"min_split_loss": 2.0347398485141888,
"max_depth": 7,
"min_child_weight": 7.552846433495573,
"max_delta_step": 3.4810667986148935,
"subsample": 0.9019590730020359,
"reg_lambda": 4.181970193287489,
"reg_alpha": 8.772118070936926,
"refresh_leaf": 0,
"grow_policy": "depthwise",
"max_leaves": 0,
"max_bin": 23,
"num_parallel_tree": 10

Podsumowanie

Odnotowaliśmy nieznaczną poprawę jakości. Niektóre wyniki były jednak słabsze niż dla początkowego modelu. Wynika to z dopasowania do zbioru danych walidacyjnych. Ma to miejsce zwłaszcza w przypadku prostszego zbioru danych. Algorytmy rzadko poprawiały jakość najlepszego punktu. Może to oznaczać, że tym sposobem nie jesteśmy w stanie osiągnąć lepszych wyników.

Wnioski końcowe

- Algorytmy heurystyczne, takie jak Przeszukiwanie Losowe (RS), Symulowane Wyżarzanie (SA) i Algorytm Ewolucyjny (EA), są przydatnymi narzędziami w

kontekście procesu strojenia hiperparametrów algorytmu XGBoost. SA, inspirowane procesem wyżarzania metali, oraz EA, oparte na mechanizmach ewolucji biologicznej, zdają się efektywnie przeszukiwać przestrzeń hiperparametrów w poszukiwaniu optymalnych konfiguracji modelu. W naszym badaniu, SA wykazało się wyższą efektywnością w kontekście konkretnego zbioru danych, przewyższając osiągi EA. To podkreśla znaczenie odpowiedniego wyboru algorytmu optymalizacji zależnego od charakterystyki danych oraz wymagań zadania.

- Inicjacja algorytmów optymalizacji zbliżonymi do domyślnych ustawień XGBoost okazała się kluczowa dla efektywności procesu strojenia. Wybór początkowych punktów w przestrzeni hiperparametrów wpływa na tempo zbieżności do optymalnego rozwiązania. Dla przypadku XGBoost, wykorzystanie wartości początkowych zbliżonych do domyślnych parametrów przyczyniło się do szybszego osiągnięcia zbieżności oraz uzyskania lepszych wyników. Jednak, równocześnie, istnieje potrzeba ostrożności w przypadku ograniczania zakresu poszukiwań, aby nie pominąć potencjalnie lepszych konfiguracji.
- W problemach klasyfikacji, wybór odpowiedniej metryki oceny modelu jest kluczowy. W naszym badaniu, zauważyliśmy, że metryka accuracy może być nieodpowiednia w sytuacjach, gdzie jedna klasa dominuje nad pozostałymi. W takich przypadkach, preferowana jest metryka Gini, która uwzględnia zbalansowanie klas i może lepiej odzwierciedlać jakość modelu. To podkreśla konieczność świadomego podejścia do wyboru metryki w zależności od charakterystyki danych.
- Wykluczenie boostera 'gblinear' przyniosło wymierne korzyści w efektywności wykorzystania zasobów obliczeniowych. Boostery w XGBoost determinują podstawowe funkcje, a wybór odpowiedniego ma kluczowe znaczenie. W praktyce, wybór odpowiedniego boostera może zależeć od charakterystyki danych oraz dostępnych zasobów obliczeniowych. Dla zbiorów danych o bardziej skomplikowanej strukturze, 'gbtree' może okazać się bardziej efektywny.
- Przestrzeń hiperparametrów XGBoost jest ogromna, co stwarza wyjątkowe wyzwania w procesie optymalizacji. Znalezienie optymalnego zestawu parametrów w tak rozbudowanej przestrzeni wymaga skomplikowanych strategii poszukiwań. Warto również zauważyć, że poszukiwania nie zawsze prowadzą do jednoznacznej poprawy wyników, co sugeruje, że pewne zestawy parametrów mogą być równie skuteczne.
- Mimo wielu możliwości dostrojenia hiperparametrów, domyślne ustawienia XGBoost wykazały się relatywnie skuteczne. To sugeruje, że nie zawsze istnieje potrzeba głębokiego strojenia każdego parametru, zwłaszcza gdy dane są wystarczająco reprezentatywne. Optymalizacja nie zawsze prowadzi do znaczącej poprawy modelu, a przesadne strojenie może prowadzić do przeuczenia.
- W kontekście dalszych badań nad optymalizacją hiperparametrów dla XGBoost, warto zwrócić uwagę na możliwości zastosowania dyskretyzacji przestrzeni poszukiwań. Dyskretyzacja polega na ograniczeniu zbioru możliwych wartości hiperparametrów do skończonego, dyskretnego zbioru. Ten kierunek badań może przynieść następujące korzyści:
    - Dyskretyzacja przestrzeni hiperparametrów może znacznie zmniejszyć liczbę możliwych kombinacji do rozważenia. Ograniczenie przestrzeni poszukiwań do dyskretnych punktów może przyspieszyć proces optymalizacji, zwłaszcza przy wykorzystaniu algorytmów heurystycznych.

- Dyskretyzacja może być użyteczna w przypadku ograniczonych zasobów obliczeniowych, ponieważ redukuje liczbę wymagających zasobów punktów w przestrzeni hiperparametrów. To szczególnie ważne w przypadku dużych i złożonych zbiorów danych.
- Dyskretyzacja może umożliwić uproszczenie algorytmu przeszukiwań. Możliwe jest zastosowanie klasycznych algorytmów operujących na liczbach dyskretnych. Nie jest konieczne rozpatrywanie ograniczeń wynikających z różnych typów parametrów.