Algorytmy ewolucyjne

Szymon Gut

May 2023

Spis treści

1 Opis projektu

W ramach tego projektu został zaimplementowany algorytm genetyczny optymalizujący wartość funkcji w R^n. W pierwszej fazie projektu wspomniany algorytm został przetestowany na pięciowymiarowej funkcji Rastrigina oraz funkcji trójwymiarowej danej wzorem: $x^2 + y^2 + 2z^2$. W dalszej części za pomocą napisanego algorytmu rozwiązano wariant problemu znanego w literaturze jako *cutting stock problem*. Problem polegał na ułożeniu prostokątów (zadanych przez trzy liczby: wysokość, szerokość oraz wartość) w kole o danym promieniu, tak aby zmaksymalizować sumę ich wartości.

2 Wstęp teoretyczny

Algorytmy ewolucyjne to oparte na heurystyce podejście do rozwiązywania problemów, których nie można łatwo rozwiązać w czasie wielomianowym, takich jak klasyczne problemy NP-trudne (np. problem komiwojażera, dynamiczny problem marszrutyzacji). Przebieg algorytmu ewolucyjnego składa się z czterech ogólnych etapów:

- krzyżowanie
- mutacja
- ewaluacja
- selekcja

Każdy z tych etapów odpowiada określonemu aspektowi doboru naturalnego. Algorytmy genetyczne podobnie jak w teorii Darwina zakładają, że silne osobniki w populacji przeżyją i będą się rozmnażać oraz mutować, podczas gdy nieprzystosowane osobniki wymrą i nie wniosą wkładu w pulę genów kolejnych pokoleń.

Rysunek 1: Ogólny schemat algorytmu ewolucyjnego (rysunek z wykładu)

2.1 Inicjalizacja (ang. *initialization*)

Aby rozpocząć działanie algorytmu potrzebne jest stworzenie populacji rozwiązań. Populacja będzie zawierała zadaną liczbę osobników. Ważnym aspektem jest to, aby była ona dostatecznie duża, gdyż reprezentuje ona pulę genów.

2.2 Krzyżowanie (ang. *crossover*)

Jest to połączenie cech dwóch losowo wybranych osobników. Proces ten odzwierciedla rozmnażanie się i przekazywanie genów wspólnych potomstwu. Krzyżowanie powinno zazwyczaj dotykać większości populacji (prawdopodobieństwo krzyżowania $\geqslant 0.7$). Ideą tego procesu jest fakt, że połączenie dobrych cech rodziców da jeszcze lepsze cechy u potomka. Przykłady krzyżowania to np. co drugi bit od drugiego rodzica, pierwsza połowa wektora od jednego, druga połowa od drugiego rodzica, średnia arytmetyczna z cech rodziców.

2.3 Mutacja (ang. *mutation*)

Odzwierciedlenie mutacji w genotypie. Wprowadza to, więc pewne losowe zaburzenie w danym genotypie co odpowiada w rzeczywistości losowym przesunięciom rozwiązań w dopuszczalnej przestrzeni poszukiwań. Z jednego osobnika powstaje zmodyfikowana wersja tego osobnika. Zazwyczaj dotyczy to małej części populacji, każdy z osobników podlega mutacji najczęściej z prawdopodobieństwem w okolicach 20%. Dzięki temu zapewniamy różnorodność naszej populacji, co pociąga za sobą eksploracje przestrzeni poszukiwań. Jest to pewien rodzaj błądzenia losowego, pozwala na odkrywanie nowych rozwiązań i przeszukiwanie nowych obszarów. Przykłady mutacji to np. odwrócenie bitów, dodanie pewnego szumu do genotypu, zamiana jednego bitu na przeciwny.

2.4 Selekcja (ang. *selection*)

Jest to wybór osobników do kolejnej generacji. Z większą szansą chcemy wybierać osobniki, które są lepiej przystosowane (te z wyższą wartością funkcji oceny). Zazwyczaj jednak każdy z osobników ma niezerową szansę przeżycia (nawet te najsłabsze). Proces ten poprzedza ewaluacja, czyli wyliczenie funkcji przystosowania każdego z osobników. Jako taktykę przy selekcji często używa się *elitaryzm*, polegający na tym że pewna liczba najlepszych osobników automatycznie (bez udziału w procesie) przechodzi do kolejnego pokolenia.

3 Pierwsza część projektu AE1

W tej części jak to zostało wcześniej wspomniane został zaimplementowany algorytm genetyczny, z mutacją Gaussowską oraz drugą polegającą na dodaniu wektora int'ów do danego genotypu zawierającego losowe elementy całkowite z przedziału (0,10). Jako krzyżowanie zostałą zaimplementowana metoda krzyżowania jednopunktowego oraz krzyżowania polegającego na wektorze średnich wartości z obu rodziców.

W dalszej części wspomniany algorytm został przetestowany na dwóch funkcjach a wyniki zostaną zaprezentowane poniżej.

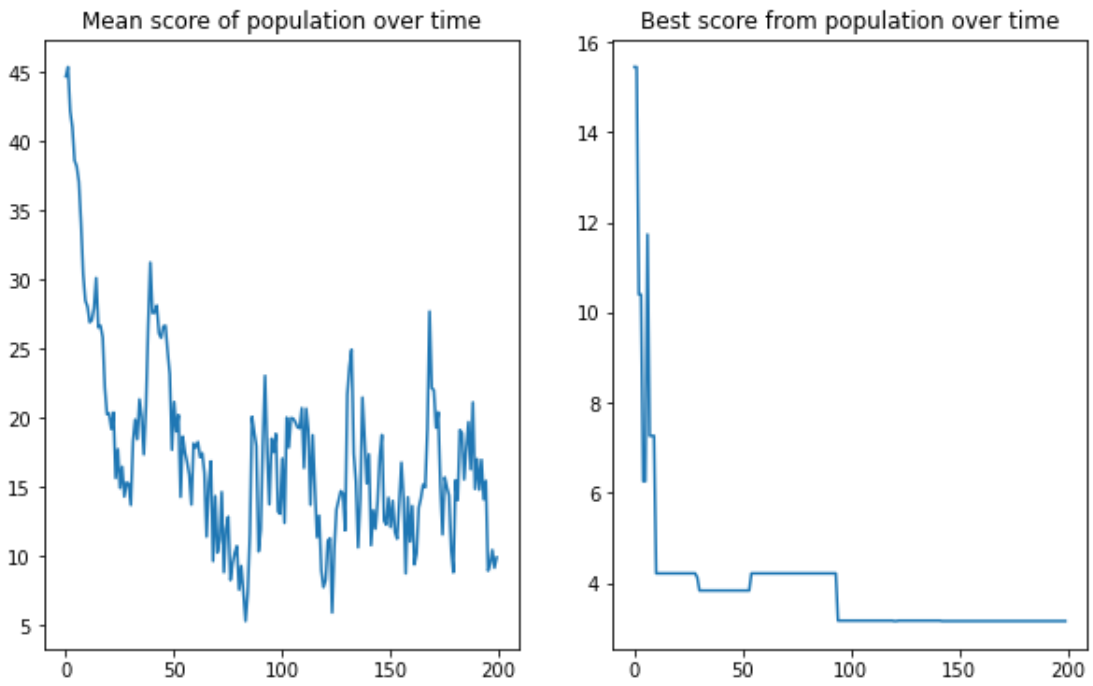
3.1 Pięciowymiarowa funkcja Rastringa

$$f(x) = An + \sum_{i=1}^{n}[x_i^2 - A\cos(2\pi x_i)] \tag{1}$$

gdzie A = 10, a n jest ustawione na 5 z faktu, że wskazane było przetestować tą funkcję w przestrzeni pięciowymiarowej.
Jeśli chodzi o wybór parametrów w algorytmie to zostały one ustawione następująco:

- *crossover_ratio*: 0.75
- *mutation_ratio*: 0.2
- *population_size*: 70
- *selection_function*: elitaryzm
- *mutation_type*: gaussowska
- *crossover_type*: krzyżowanie jednopunktowe
- *iterations*: 200

Jeśli chodzi o najlepszy wynik jaki udało się osiągnąć podczas minimalizacji tej funkcji to wynosił on 3.16.
Wartość tą udało się uzyskać dla $\mathbf{x}$=[-0.04614599; 0.01655535; -0.08164264; 0.02982952; 1.02857832]
Średni wynik dla całej populacji to 9.9.
Poniżej przedstawiono wykres prezentujący średnie wyniki dla całej populacji oraz wartość najlepszego osobnika w danej populacji w zależności od kolejnych iteracji.



Rysunek 2: Przebieg algorytmu ewolucyjnego dla funkcji Rastringa

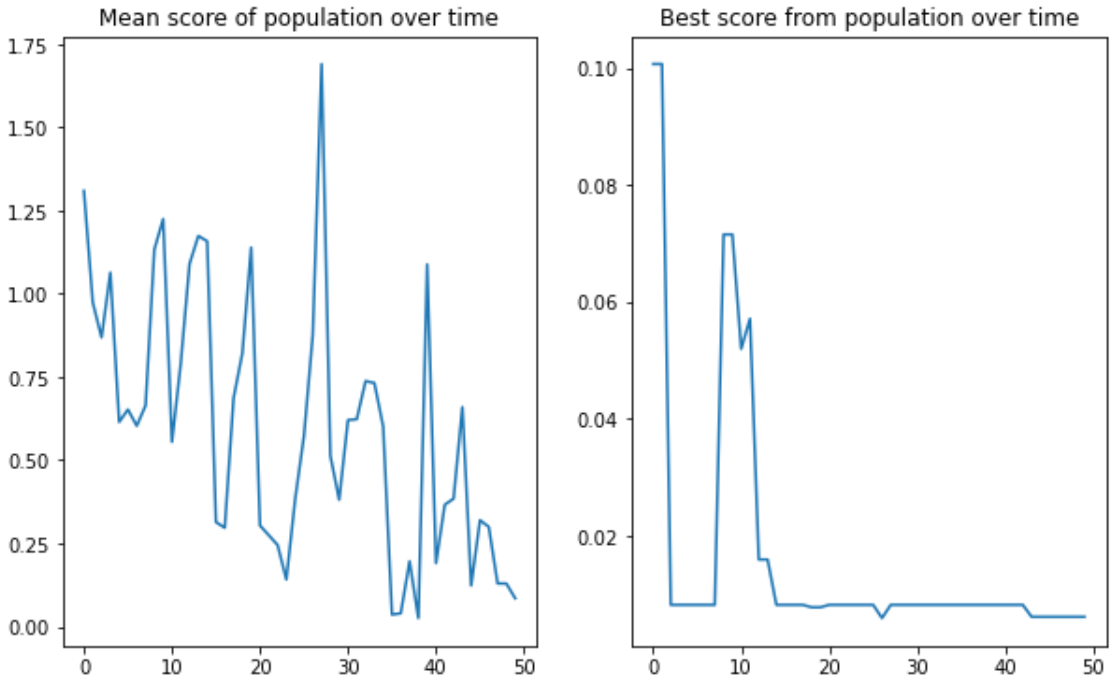
3.2 Trójwymiarowa funkcja

$$f(x, y, z) = x^2 + y^2 + 2z^2 \tag{2}$$

W dalszej części minimalizowano funkcję zadaną wzorem (2).
Jeśli chodzi o wybór parametrów w algorytmie to zostały one ustawione następująco:

- *crossover_ratio*: 0.8
- *mutation_ratio*: 0.2
- *population_size*: 40
- *selection_function*: elitaryzm
- *mutation_type*: gaussowska
- *crossover_type*: krzyżowanie jednopunktowe
- *iterations*: 50

Najlepszy wynik jaki udało się uzyskać minimalizując powyższą funkcję wynosił 0.006.
Wartość ta została uzyskana dla **x**=[-0.03751747; 0.02945818; 0.04424433].
Średni wynik dla całej populacji wynosił 0.085.
Poniżej przedstawiono wykres prezentujący średnie wyniki dla całej populacji oraz wartość najlepszego osobnika w danej populacji w zależności od kolejnych iteracji.



Rysunek 3: Przebieg algorytmu ewolucyjnego dla funkcji trójwymiarowej

3.3 Podsumowanie

W celu lepszej minimalizacji funkcji, możnaby wypróbować zaimplementowany algorytm ustawiając większą liczbę iteracji, minimalnie modyfikować hiperparametry oraz wypróbować również inną regułę mutacji, czy krzyżowania.

Wynik jednak można uznać za zadowalający, gdyż uzyskana wartość dla funkcji trójwymiarowej jest bardzo bliska 0, a jest to minimum globalne tej funkcji w dziedzinie liczb rzeczywistych.

Patrząc na funkcję Rastrigina, jest to funkcja zdecydowanie cięższa to zminimalizowania, ze względu na mnogość minimów lokalnych tej funkcji. Wartości minimalne różnią się od zera w przypadku losowego wyboru początkowych wartości zmiennych. Wynik 3.16 choć odbiega od rzeczywistego minimum globalnego równego 0, to nadal można go uznać za zadowalający.

4 Druga część projektu AE2

W drugiej części projektu, celem było rozwiązanie problemu zwanego w literaturze pod nazwą *cutting stock problem*.

W tym celu rozważono dodatkową procedurę krzyżowania - *mean_crossover* polegającą na wyliczeniu wektora średnich cech z obu rodziców. Użyta mutacja wprowadzała losowe zaburzenie z przedziału (-5, 5) do chromosomu. Gdy w nowo uzyskanym chromosomie, któryś z elementów stawał się liczbą ujemną, zastępowano go 0 (nie możemy mieć ujemnej liczby prosotkątów).

Wyniki eksperymentu należało sprawdzić na 5 zbiorach danych:

- r800 - koło o średnicy 800
- r850 - koło o średnicy 850
- r1000 - koło o średnicy 1000
- r1100 - koło o średnicy 1100
- r1200 - koło o średnicy 1200

Każdy z tych zbiorów zawiera wymiary prostokątów oraz ich wartość.
Cel to zmaksymalizowanie wartości, to znaczy wybór takich prostokątów do wpisania w ten okrąg aby ich sumaryczna wartość była największa.

Ważnym etapem było wyznaczenie funkcji nagradzającej dane osobniki w populacji i karającej te, które zwracają niekorzystne lub zakazane wyniki. Tak, więc aby karać osobniki wykonujące ruchy zakazane, to jest zwracające zbyt dużą liczbę prostokątów danych wymiarów przez co nie

mieszczą się one w okręgu, funkcja oceny przypisuje im wartość równą 0. Jeśli dany osobnik z populacji przestrzega zasady gry, zwraca dozwolone prostokąty, tak że mieszczą się one w zadanym okręgu, przypisujemy im wartość równą sumarycznej wartości zwróconej przez nie prostokątów.

Celem jest maksymalizowanie funkcji oceny, to jest aby umieścić w okręgu takie prostokąty, których wartość sumaryczna będzie największa.

4.1 Dokładny opis algorytmu

Poniżej zostaną przedstawione szczegóły implementacji oraz kodowania tego problemu. Jest to jedyna sekcja, gdzie zdecydowano się zamieścić parę fragmentów kodu, jednakże jest to spowodowane trudnością samej tej części projektu i może być to niezbędne do zrozumienia rozwiązania.

4.1.1 Opis kodowania

Osobnik w populacji jest kodowany jako wektor zawierający ilość prostokątów poszczególnych wymiarów. To znaczy, że jeśli mamy prostokąty o wymiarach 10x120, 120x10, 400x20, 300x30 i osobnika w populacji [100, 10, 50, 4] to koduje on ilość prostokątów danych wymiarów (np. 100 prostokątów o wymiarach 10x120).

Funkcja oceny osobnika przydziela sumaryczną wartość prostokątów jakie udało się wpisać w okrąg. Jeśli któryś z osobników nie spełnił zasad gry, tj. posiada zbyt dużą ilość prostokątów przez to te nie mieszczą się w okręgu, funkcja oceny przypisuje im wartość 0.

Przechodząc do kluczowej chwili w tym algorytmie tj. umieszczania prostokątów w okręgu, to są one od umieszczne od góry okręgu i gdy nie mieszczą się już w danym wierszu, algorytm przechodzi do kolejnego wiersza i w nim umieszcza kolejne prostokąty. Prostokąty zawsze układane są poziomo tj. "leżą one na swojej dłuższej krawędzi" i są układane jeden przy drugim. Prostokąty zaczynają być układane od lewej strony w każdym wierszu, a pozycje w których możemy zacząć układać kolejny rząd są walidowane przez funkcje fit_horizontal oraz fit_left. Funkcje sprawdzają czy prostokąty mieszczą się jeszcze w okręgu, jeśli nie to przechodzimy do kolejnego wiersza.

Dokładny opis kodowania rozwiązań wyszczególniony w oddzielnych krokach:

1. Pierwszy prostokąt umieszczany jest na samej górze okręgu na śródku, aby zmaksymalizować użyte miejsce.

```
y = pitagoras(rectangles[0].width / 2, None, radius)

rectangles[0].position = (
-rectangles[0].width / 2, y - rectangles[0].height)
```

2. Stworzenie listy current_row przechowującej prostokąty w danym wierszu.

3. Rozpoczęcie iteracji po pozostałych prostokątach (po indeksach od 1 do końca listy).

4. Dla każdego prostokąta sprawdzane jest czy można go umieścić na tej samej linii co poprzedni prostokąt (poziomo). W przypadku gdy szerokość poprzedniego prostokąta pozwala na umieszczenie obecnego prostokąta, obliczana jest standardowa pozycja na podstawie ułożenia wcześniejszego sąsiadującego prostokąta (prev_upper_right_corner[0], prev_upper_right_corner[1] – rect.height) gdzie indeksy w prawym górnym rogu oznaczają odpowiednio współrzędną x-ową oraz y-ową.

5. Jeśli obecny prostokąt nie może zostać umieszczony na tej samej linii co poprzedni prostokąt, obliczana jest maksymalna wysokość poprzedniego rzędu prostokątów.

6. Tworzony jest nowy rząd prostokątów, a pozycja obecnego prostokąta obliczana jest na podstawie maksymalnej wysokości poprzedniego rzędu i położenia wcześniejszego prostokąta. Metoda wygląda następująco

```
def calculate_position_in_new_row(prev_row_max_height, prev_rect, rect,
                                  radius):
    rect_upper_y = prev_rect.position[
                       1] + prev_rect.height - prev_row_max_height
    rect_left_x = -pitagoras(rect_upper_y, None, radius)

    rect_lower_y = rect_upper_y - rect.height
    if is_in_circle(rect_left_x, rect_upper_y, radius) and is_in_circle(
```

```
            rect_left_x, rect_lower_y, radius):
        return rect_left_x, rect_lower_y
    else:
        return max(-pitagoras(rect_upper_y, None, radius),
                   -pitagoras(rect_lower_y, None, radius)), rect_lower_y
```

7. Obecny prostokąt dodawany jest do listy current row.
8. Proces jest powtarzany dla wszystkich prostokątów

4.1.2 Opis algorytmu

Główną logikę stanowi kodowanie przedstawione w sekcji pierwszej. Jeśli chodzi o uniwersalny algorytm ewolucyjny to nie odbiega on niczym od standardowych algorytmów ewolucyjnych. Tworzona jest populacja, następnie z ustalonym p-stwem odbywa się krzyżowanie, następnie również z ustalonym p-stwem odbywa się mutacja, następnie generowane są wyniki oceny poszczególnych osobników oraz odbywa się selekcja do nowej generacji. Proces ten jest powtarzany określoną liczbę razy (ilość ustawionych iteracji).

Zostało zaimplementowane kilka metod mutacji jednak ta używana w tej pracy domowej to **int mutation**. Mutacja ta dodaje losowe zaburzenie w postaci liczb całkowitych do chromosomu. Jeśli jednak na którymś miejscu w tablicy uzyskamy wartość ujemną, podstawiamy tam 0 (nie może być ujemnej liczby prostokątów!).

```
def int_mutation(chromosome):

    vector_to_add = np.random.randint(-5, 5, chromosome.shape[0])
    mutated = vector_to_add + chromosome
    return np.where(mutated < 0, 0, mutated)
```

Zostało zaimplementowane również kilka metod krzyżowania jednak metoda użyta w tej pracy domowej to **mean crossover**.

```
def mean_crossover(parentA, parentB):
    return (parentA + parentB) // 2
```

Do wyboru nowych populacji podczas kolejnych generacji algorytmu została użyta **elite selection** tj. wybór najlepszych osobników w populacji.

```
def elite_selection(scores, n):
    return np.argpartition(scores, -n)[-n:]
```

Powyższa logika została użyta do przeprowadzenia wszystkich eksperymentów z tej sekcji przedstawionych niżej.

4.2 Cutting Stock Problem

Dla każdego ze zbiorów danych, wskazanych do tego problemu rozpatrzono architektury z nastepującymi parametrami:

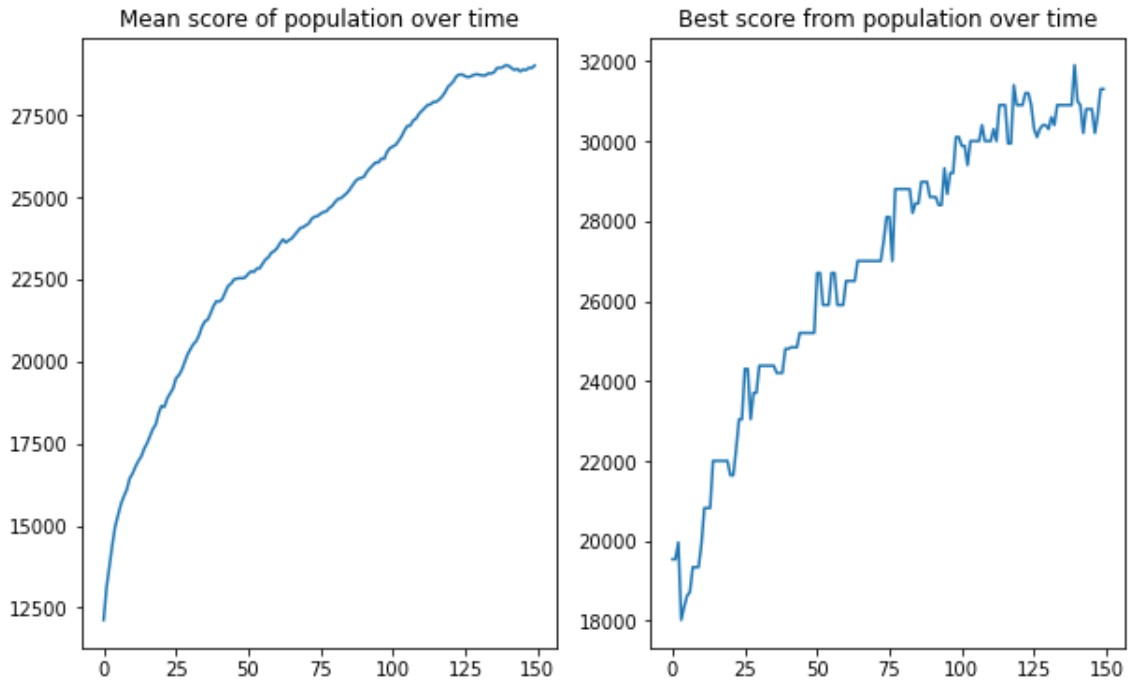
- *crossover_ratio*: 0.7
- *mutation_ratio*: 0.2
- *crossover_type*: mean_crossover
- *mutation_type*: int_mutation
- *selection_function*: elite selection

Algorytmy różniły się tylko licznością populacji oraz liczbą iteracji na jakie zostały uruchomione. Te parametry zostaną podane w odpowiednich sekcjach dla każdego ze zbiorów danych.

4.2.1 Zbiór z promieniem równym 800

Dla tego zbioru danych algorytm został uruchomiony z populacją wielkości 100 oraz liczba iteracji została ustawiona na 150.
Po 150 iteracjach algorytm dla najlepszego osobnika uzyskał wartość 31 300, czyli wartość wyższą, od wartości progowej wskazanej w poleceniu równej **30 000**.
Średni wynik całej populacji po wszystkich iteracjach wynosiłą 29 002.4.

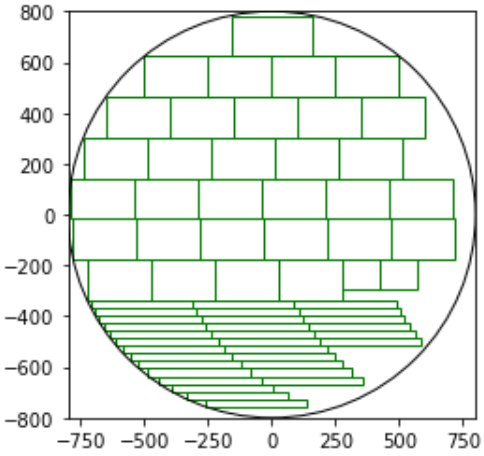
W celu jeszcze lepszej optymalizacji warto ustawić algorytm na większą liczbe iteracji lub zwiększyć liczność populacji. Nie było to jednak konieczne gdyż wynik uzyskany na wskazanych parametrach okazał się już być dostatecznie wysoki.



Rysunek 4: Przebieg algorytmu ewolucyjnego dla zbioru danych z promieniem równym 800

Najlepszy wynik został zwrócony dla następującej liczby poszczególnych prostokątów:

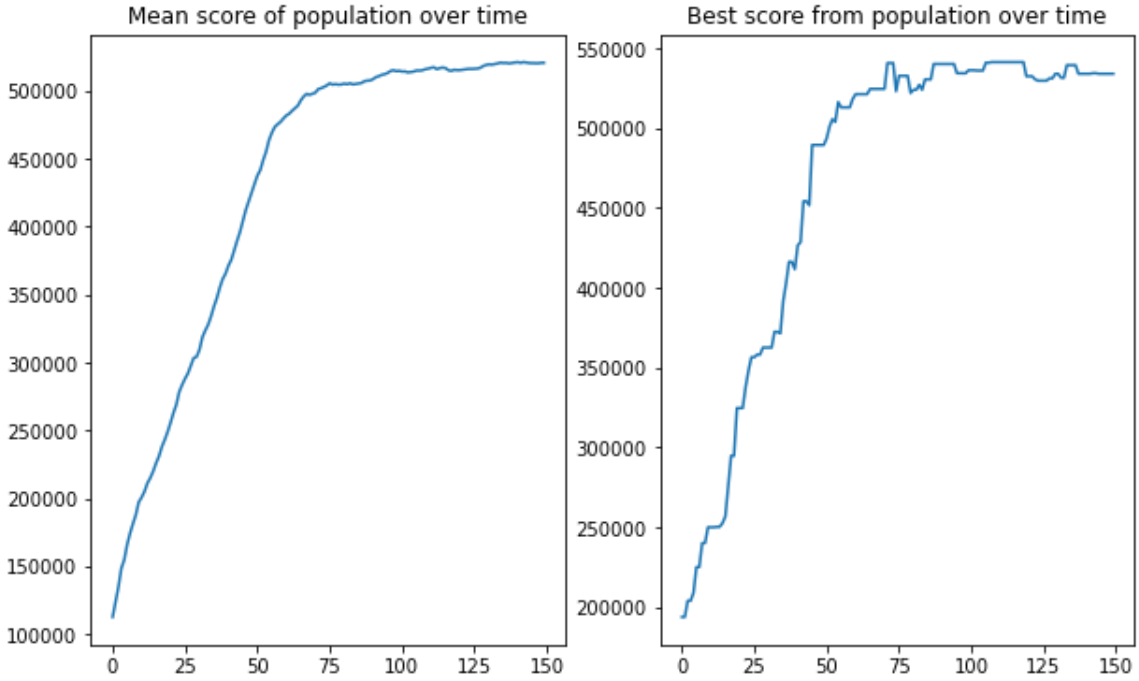
- 0 prostokątów o wymiarach 250 x 120 i wartości równej 100
- 1 prostokąt o wymiarach 320 x 160 i wartości równej 500
- 30 prostokątów o wymiarach 250 x 160 i wartości równej 600
- 0 prostokątów o wymiarach 150 x 120 i wartości równej 40
- 32 prostokąty o wymiarach 400 x 30 i wartości równej 400



Rysunek 5: Wizualizacja otrzymanych wyników na zbiorze z promieniem równym 800

4.2.2 Zbiór z promieniem równym 850

Dla tego zbioru danych algorytm został uruchomiony z populacją wielkości 60 oraz liczba iteracji została ustawiona na 150.
Po 150 iteracjach algorytm dla najlepszego osobnika uzyskał wartość 534 120.
Średni wynik całej populacji po wszystkich iteracjach wynosiłą 520 651.5.

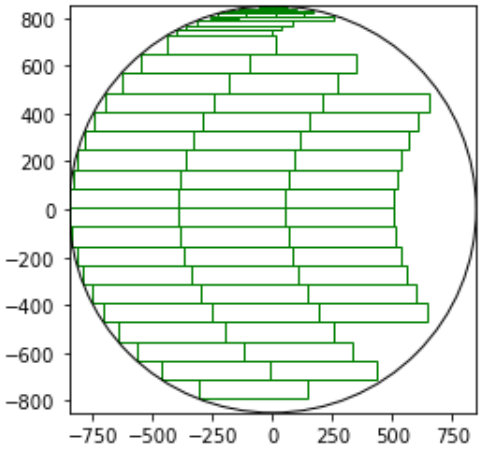


Rysunek 6: Przebieg algorytmu ewolucyjnego dla zbioru danych z promieniem równym 850

Najlepszy wynik został zwrócony dla następującej liczby poszczególnych prostokątów:

- 6 prostokątów o wymiarach 10 x 120 i wartości równej 120
- 4 prostokąty o wymiarach 120 x 10 i wartości równej 150
- 4 prostokąty o wymiarach 400 x 20 i wartości równej 1200
- 0 prostokątów o wymiarach 300 x 30 i wartości równej 1200
- 0 prostokątów o wymiarach 120 x 120 i wartości równej 1200
- 0 prostokątów o wymiarach 100 x 100 i wartości równej 900

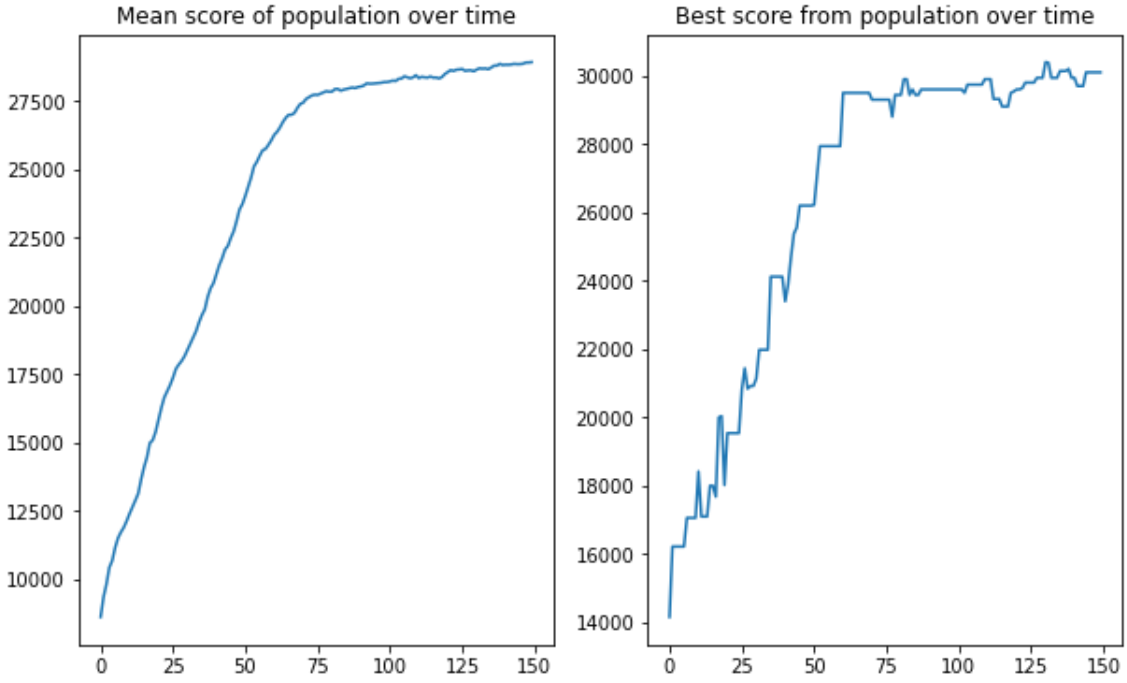
- 48 prostokątów o wymiarach 450 x 80 i wartości równej 11000



Rysunek 7: Wizualizacja otrzymanych wyników na zbiorze z promieniem równym 850

4.2.3 Zbiór z promieniem równym 1000

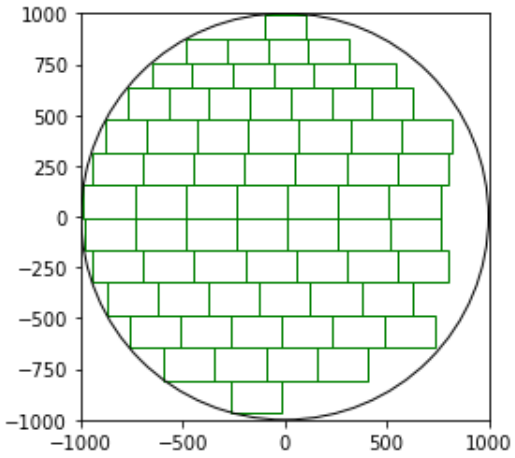
Dla tego zbioru danych algorytm został uruchomiony z populacją wielkości 60 oraz liczba iteracji została ustawiona na 150.
Po 150 iteracjach algorytm dla najlepszego osobnika uzyskał wartość 30 100 czyli wartość wyższą, od wartości progowej wskazanej w poleceniu równej **17 500**.
Średni wynik całej populacji po wszystkich iteracjach wynosiłą 28 918.67.



Rysunek 8: Przebieg algorytmu ewolucyjnego dla zbioru danych z promieniem równym 1000

Najlepszy wynik został zwrócony dla następującej liczby poszczególnych prostokątów:

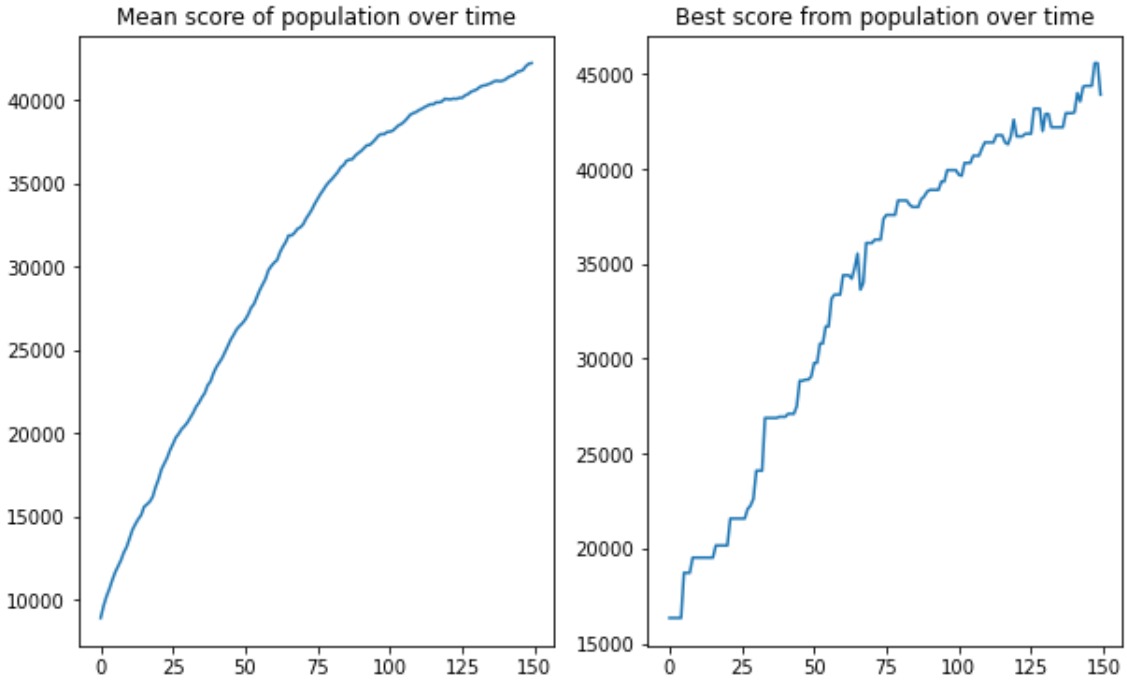
- 11 prostokątów o wymiarach 200 x 120 i wartości równej 200
- 8 prostokątów o wymiarach 200 x 160 i wartości równej 300
- 51 prostokątów o wymiarach 250 x 160 i wartości równej 500
- 0 prostokątów o wymiarach 100 x 120 i wartości równej 40



Rysunek 9: Wizualizacja otrzymanych wyników na zbiorze z promieniem równym 1000

4.2.4 Zbiór z promieniem równym 1100

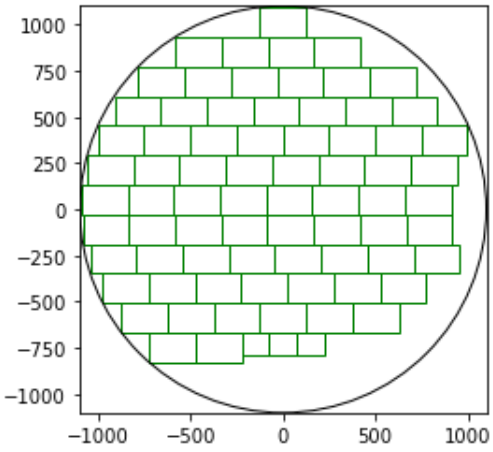
Dla tego zbioru danych algorytm został uruchomiony z populacją wielkości 60 oraz liczba iteracji została ustawiona na 150.
Po 150 iteracjach algorytm dla najlepszego osobnika uzyskał wartość 43 920, czyli wartość wyższą, od wartości progowej wskazanej w poleceniu równej **25 000**.
Średni wynik całej populacji po wszystkich iteracjach wynosiłą 42 219.67.



Rysunek 10: Przebieg algorytmu ewolucyjnego dla zbioru danych z promieniem równym 1100

Najlepszy wynik został zwrócony dla następującej liczby poszczególnych prostokątów:

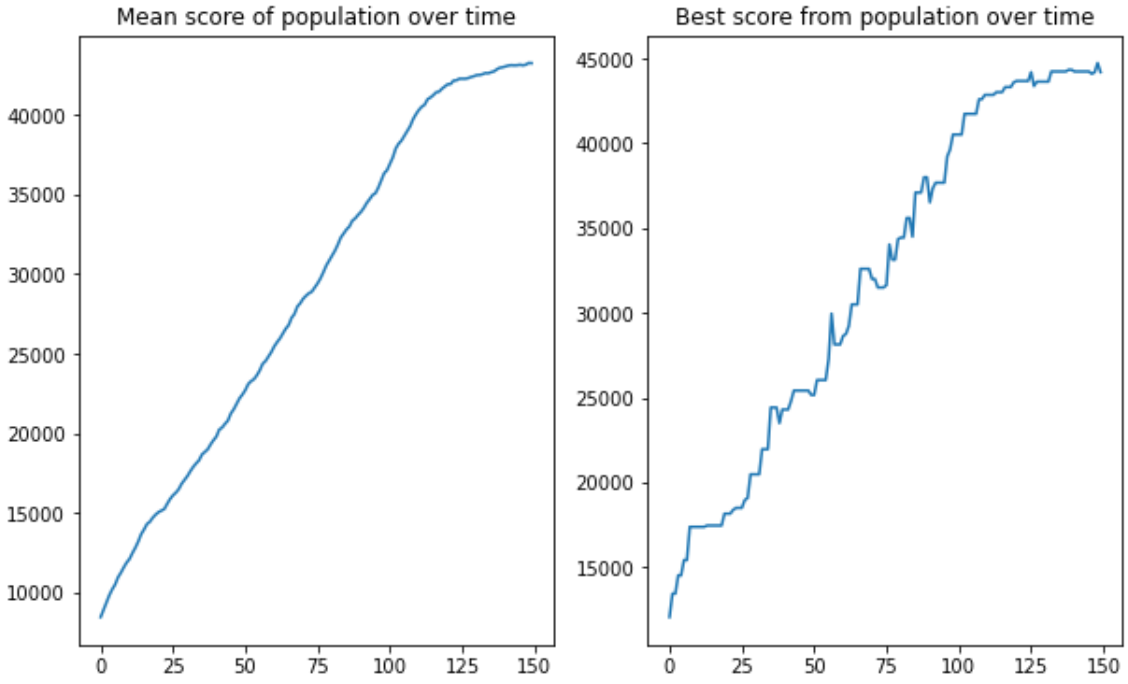
- 0 prostokątów o wymiarach 250 x 120 i wartości równej 100
- 0 prostokątów o wymiarach 120 x 360 i wartości równej 300
- 73 prostokąty o wymiarach 250 x 160 i wartości równej 600
- 3 prostokąty o wymiarach 150 x 120 i wartości równej 40



Rysunek 11: Wizualizacja otrzymanych wyników na zbiorze z promieniem równym 1100

4.2.5 Zbiór z promieniem równym 1200

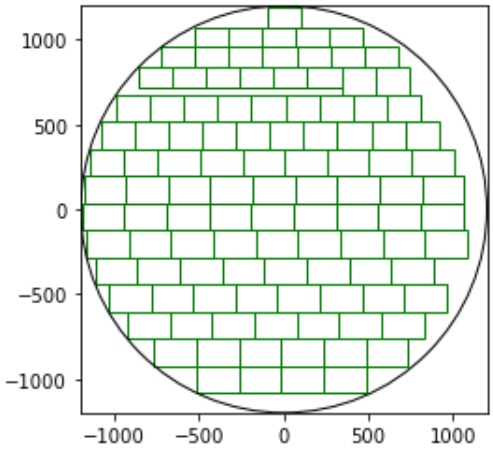
Dla tego zbioru danych algorytm został uruchomiony z populacją wielkości 60 oraz liczba iteracji została ustawiona na 150.
Po 150 iteracjach algorytm dla najlepszego osobnika uzyskał wartość 44 200, czyli wartość wyższą, od wartości progowej wskazanej w poleceniu równej **30 000**.
Średni wynik całej populacji po wszystkich iteracjach wynosiłą 43 240.



Rysunek 12: Przebieg algorytmu ewolucyjnego dla zbioru danych z promieniem równym 1200

Najlepszy wynik został zwrócony dla następującej liczby poszczególnych prostokątów:

- 19 prostokątów o wymiarach 200 x 120 i wartości równej 200
- 23 prostokąty o wymiarach 200 x 160 i wartości równej 300
- 67 prostokątów o wymiarach 250 x 160 i wartości równej 500
- 0 prostokątów o wymiarach 100 x 120 i wartości równej 40



Rysunek 13: Wizualizacja otrzymanych wyników na zbiorze z promieniem równym 1200

4.3 Podsumowanie

W ramach tej pracy domowej przetestowano algorytm ewolucyjny na 5 zbiorach. Na każdym z nich uzyskano wyniki **wyższe** od wartości wymaganych. W tym celu napisano funkcje oceny nagradzającą osobniki silne w populacji oraz karzącą osobniki słabe. Do przeprowadzenia mutacji oraz krzyzowania również skorzystano z własnych pomysłów, odmiennych niż te zaproponowane w pracy domowej pierszej (AE1).

5 Trzecia część projektu AE3

W trzeciej części projektu skupiono się na optymalizacji sieci neuronowych za pomocą algorytmów ewolucyjnych. Wyniki należało przetestować na 3 zbiorach danych:

- multimodal-large
- auto-mpg
- iris

Podczas implementacji skorzystano z klasy perceptronu stworzonej w ramach realizowania przedmiotu w wcześniejszej fazie semestru. Jako osobniki w populacji przyjmujemy wagi sieci neuronowych, inicjowane na początku z rozkładu jednostajnego na przedziale [-1, 1].

Jako metoda selekcji została zastosowana selekcja elitarna, tzncz że do kolejnych generacji wybierane są tylko najsilniejsze osobniki. Zadaniem populacji jest minimalizowanie funkcji straty, to jest MSE (Mean Squared Error) w przypadku regresji lub Cross-Entropy w przypadku klasyfikacji.

Podczas przeprowadzania eksperymentów bardzo ciężko było zredykować wartość funkcji straty, porównując to do czasu jaki potrzebowaliśmy żeby tego dokonać używając mechanizmu propagacji wstecznej. Algorytm mimo, że minimalnie minimalizował funkcję straty to generował to bardzo małymi krokami.

Wnioski jakie nasuwają się po tych eksperymentach, to fakt że algorytmy ewolucyjne mimo że są zdecydowanie łatwiejsze do zrozumienia i nie wymagają aż takiej wiedzy matematycznej jak chociażby w przypadku algorytmu propagacji wstecznej potrzebują bardzo wielu zasobów aby uzyskać zadowalający wynik oraz nie uzyskają tak dobrych rezultatów jak standardowa procedura uczenia.

Jeśli chodzi o zastosowany mechanizm krzyżowania to polegał on na losowej wymianie warstw pomiędzy rodzicami.

Przechodząc natomiast do mutacji to zastosowano tutaj wprowadzanie szumu gaussowskiego, aby minimalnie zmienić wagi modelu a jednocześnie nie wychodzić poza zakres [-1,1].

5.1 Multimodal-Large

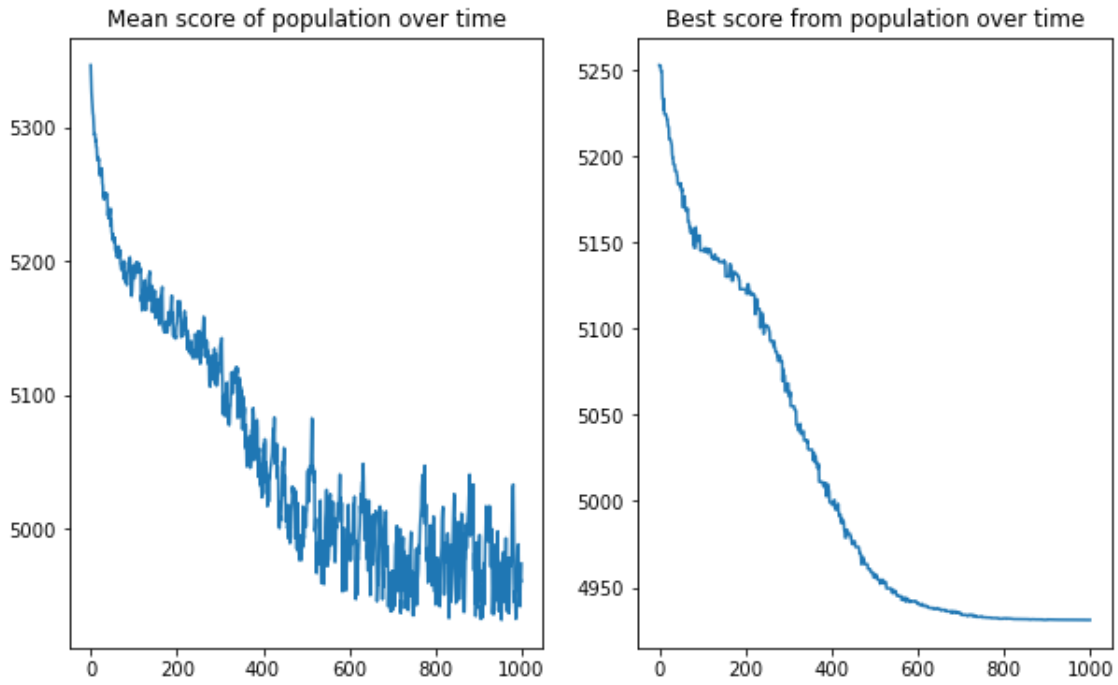
W pierwszym eksperymencie przeprowadzono testy na zbiorze **multimodal-large**. Mimo, że MSE udało się znacznie zredukować (drugi eksperyment w tej sekcji) to porównując je do tego uzyskanego przy propagacji wstecznej, nie można uznać ich za satysfakcjonujące.

Pierwszy eksperyment doskonale pokazuje, że metoda mutacji oraz krzyżowania bardzo mocno wpływa na proces ewolucji osobników. W pierwszym eksperymencie ustawiono mechanizm krzyżowania polegający na losowym przydzielaniu wag w odniesieniu do rodziców. Nie był to najlepszy sposób, gdyż zmieniając pojedyńcze wagi w zależności od rodzica wprowadzamy bardzo duży chaos w warstwach sieci i nie poprawia to praktycznie minimalizowanego MSE. Jeśli chodzi o zastosowaną mutację w tej metodzie to została zastosowana mutacja maskowa polegająca na początku na przypisaniu losowej maski boolowskiej i względem niej były przydzielane lub nie przydzielane losowe zaburzenia.

Ustawiając nastepujące parametry modelu:

- crossover_ratio = 0.8
- mutation_ratio = 0.15
- crossover = one_point_multi_dim
- mutation = mask_mutation
- selection_func = elite_selection
- population_size = 1000
- input_shape_network=x_train_modal.shape
- neurons_num=[15,15,1]
- activations=[Sigmoid(), Sigmoid(), Linear()]
- X = x_train_modal
- y = y_train_modal
- loss = Mse()

Ustawiając liczbę iteracji na 1000, **MSE** jakie udało się ostatecznie uzyskać było równe **4973**. Nie można tutaj nawet mówić o uczeniu się sieci gdyż startowym poziomem MSE na tym zbiorze było **5253**.



Rysunek 14: Wizualizacja otrzymanych wyników na zbiorze multimodal-large, z metodą losowego, pojedyńczego przydzielania wag od rodziców

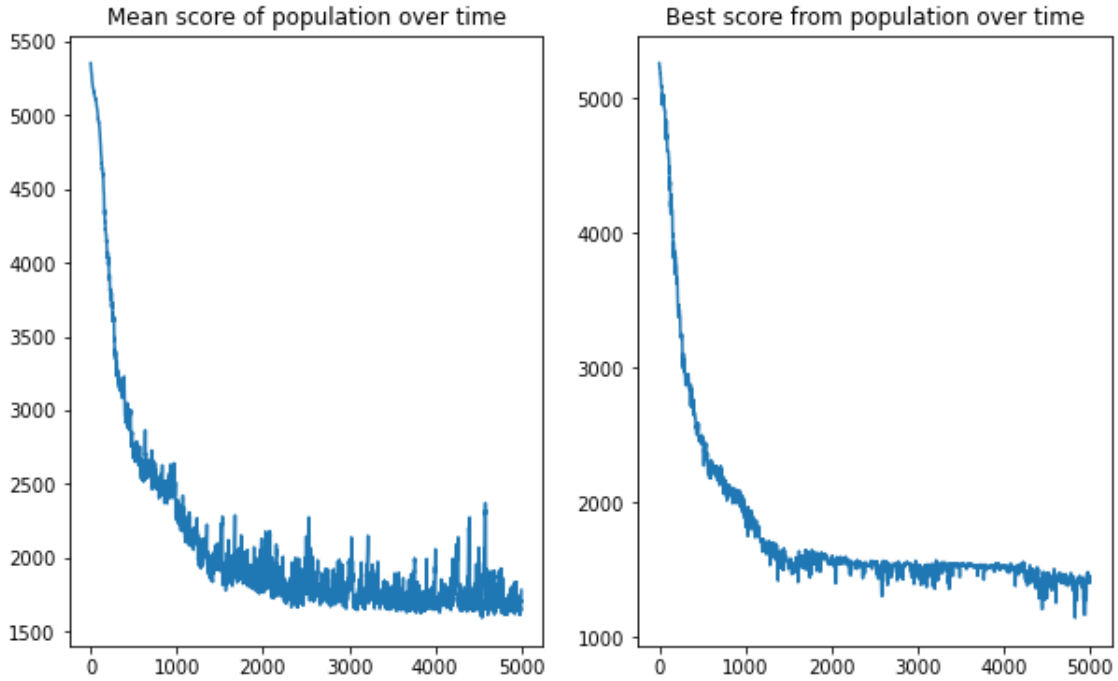
Dalej zastąpiono metodę krzyżowania oraz mutacji, gdyż ewidentnie nie spełniały one założonych wymagań. Tym razem podczas krzyżowania osobników, ich potomek miał już nie losowo

przydzielane każde z wag, jednak była już tu losowo przydzielana cała warstwa rodzica. Jeśli chodzi o mutacje, dodawana była macierz szumu gaussowskiego w celu lepszej eksploracji rozpatrywanej przestrzeni wag.

Jeśli chodzi o rozpatrywane parametry algorytmu prezentowały się one następująco:

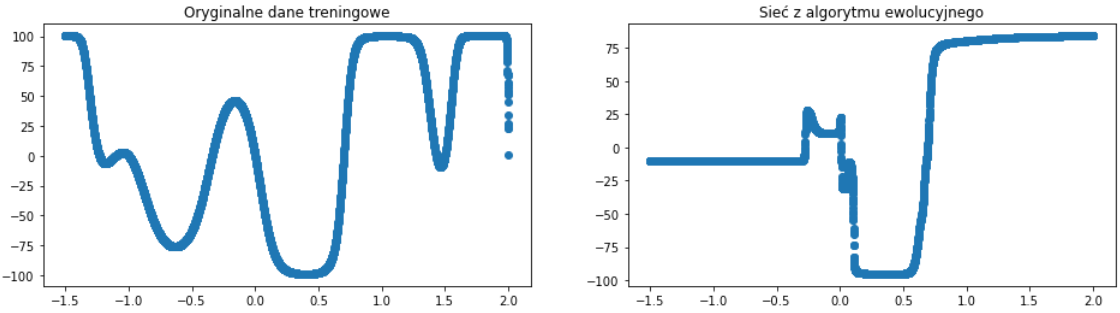
- crossover_ratio = 0.7
- mutation_ratio = 0.2
- crossover = random_layer
- mutation = noise_mutation
- selection_func = elite_selection
- population_size = 100
- input_shape_network=x_train_modal.shape
- neurons_num=[10,10,10,1]
- activations=[Sigmoid(), Sigmoid(), Sigmoid(), Linear()]
- X = x_train_modal
- y = y_train_modal
- loss = Mse()

Rezultaty dla tak dobranych parametrów ewolucji przedstawiają się już znacznie inaczej. Minimalizacja funkcji straty jest już jak najbardziej zauważalna, jednakże nadal ma się ona nijak w porównaniu do mechanizmu propagacji wstecznej. Uzyskane MSE po 5000 iteracji wynosiło około **1450**, podczas gdy testowanie podczas laboratoriów z sieci neuronowych pokazało, że możliwe jest uzyskanie nawet MSE na poziomie **40** lub niższym.

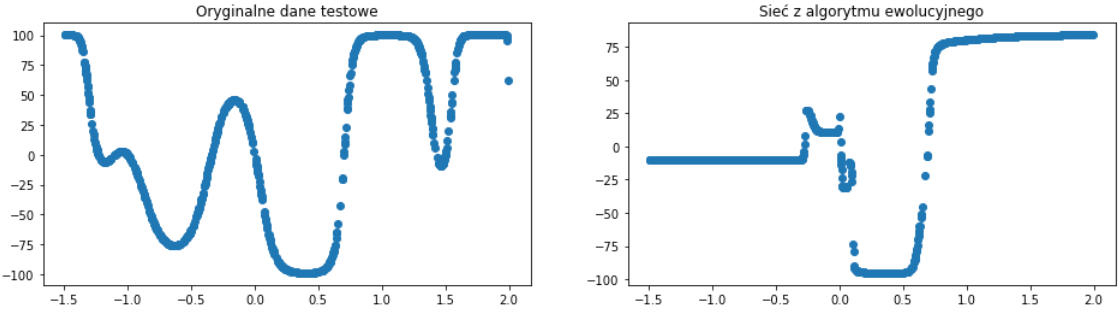


Rysunek 15: Wizualizacja otrzymanych wyników na zbiorze multimodal-large, z metodą przydzielania całej, losowej warstwy rodzica

Mimo, że MSE zostało w znaczny sposób zredukowane to jakość modelu idealnie ukazują poniższe wizualizacje.

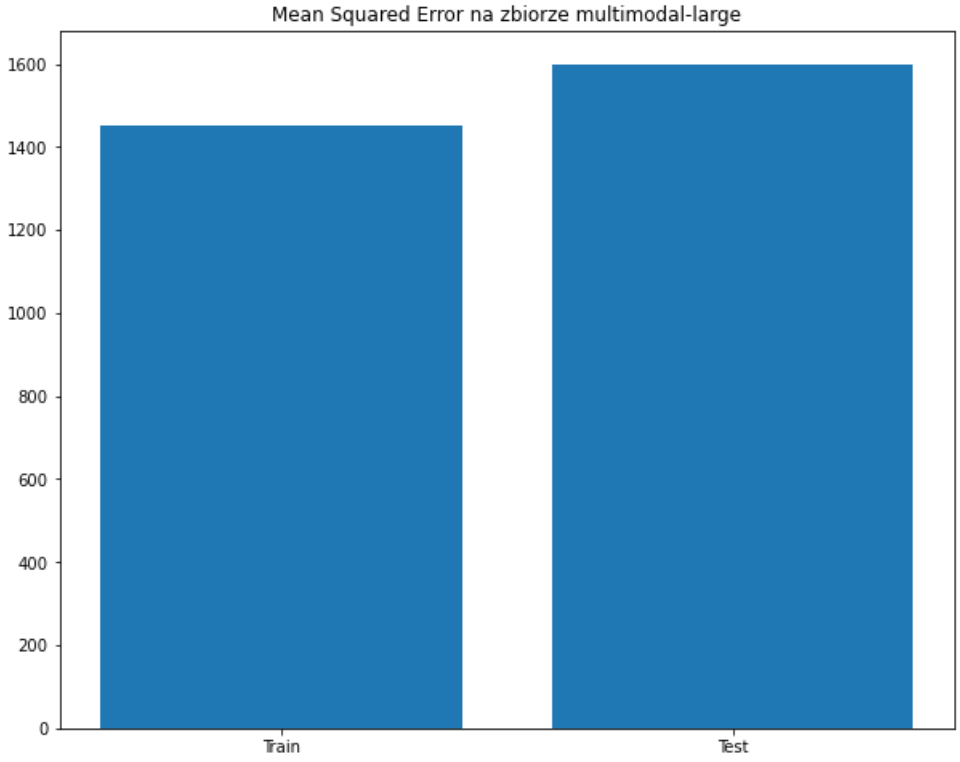


Rysunek 16: Wizualizacja predykcji modelu na zbiorze treningowym



Rysunek 17: Wizualizacja predykcji modelu na zbiorze testowym

MSE uzyskane na zbiorze treningowym było równe **1452**. Jeśli chodzi natomiast o zbiór testowy to wynosiło ono **1599**.



Rysunek 18: Porównanie uzyskanego MSE na zbiorze multimodal-large

5.2 Zbiór iris

Kolejnym zbiorem użytym do testowania był zbiór iris. Zadaniem do zrealizowania była klasyfikacja irysa, ze względu na różne cechy kwiatów takie jak długość płatka kwiatu, szerokość płatka, długość

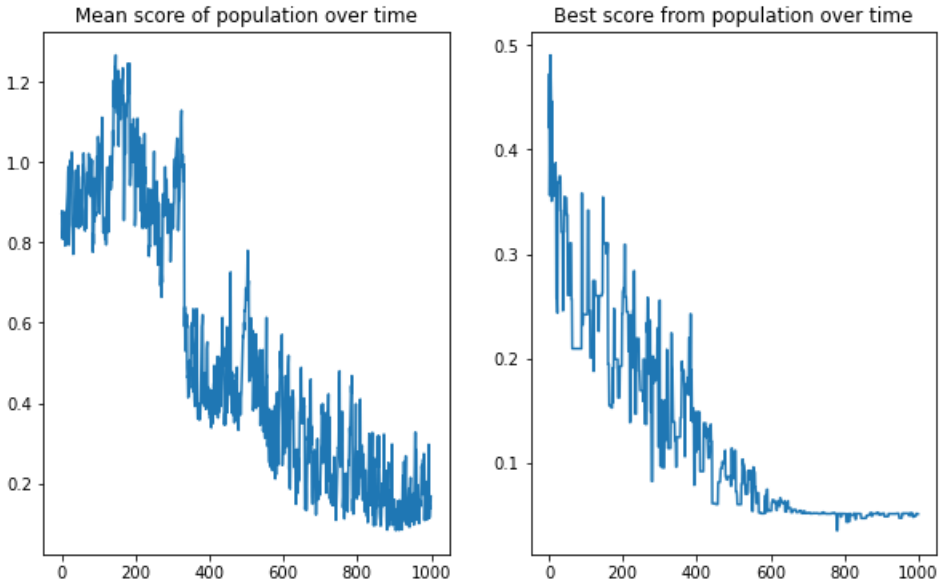
działki kielicha itp.

Zbiór ten z uwagi na swoją prostotę pozwolił wytrenować sieci na bardzo wysokim poziomie. Zanim zaczęto trenowanie sieci podzielono zbiór iris na zbiór testowy oraz treningowy. Podział ten przyda się do późniejszego sprawdzania jakości modelu.

Ustawione parametry algorytmu:

- crossover_ratio = 0.7
- mutation_ratio = 0.2
- crossover = random_layer
- mutation = noise_mutation
- selection_func = elite_selection
- population_size = 200
- input_shape_network=x_iris.shape
- neurons_num=[10,10,3]
- activations=[Tanh(), Tanh(), Softmax()]
- X = X_iris_train
- y = y_iris_train
- loss = Cross_entropy()

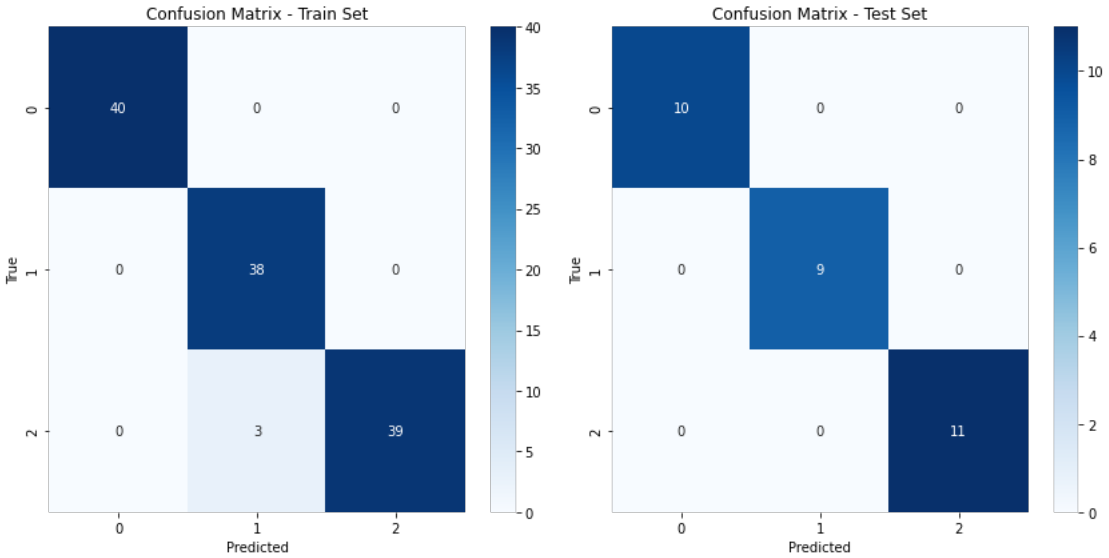
Algorytm ustawiony na 1000 iteracji zapewnił nam wartość Cross Entropii dla najlepszego osobnika na poziomie **0.05**.



Rysunek 19: proces ewolucji na zbiorze iris

Przechodząc do testowania i porównywania wyników uzyskano f1 score na poziomie **0.975** na zbiorze treningowym oraz **1.0** na zbiorze testowym.

Przechodząc natomiast do macierzy konfuzji:



Rysunek 20: macierz konfuzji dla zbioru treningowego oraz testowego

5.3 Zbiór auto-mpg

Ostatnim testowanym zbiorem był zbiór auto-mpg. Rozważany był tu problem regresji, a przewidywaną zmienną była wartość mpg.

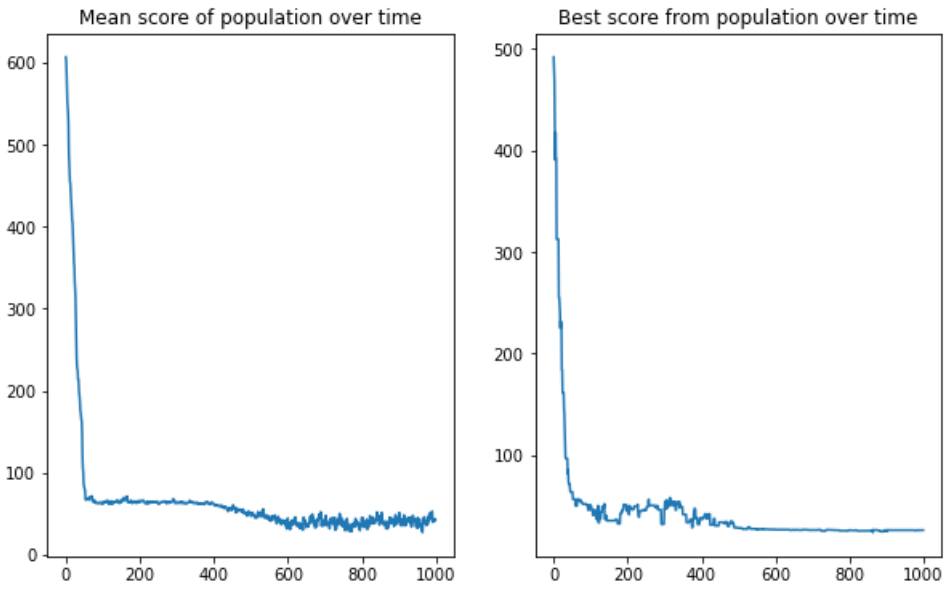
W wybranym zbiorze danych została usunięta kolumna zawierająca marki samochodów oraz zostały usunięte wartości brakujące oznaczone jako "?" w tym zbiorze danych.

W dalszej cześci podzielono zbiór na treningowy oraz testowy jak podczas wcześniejszego eksperymentu i przetestowano algorytm.

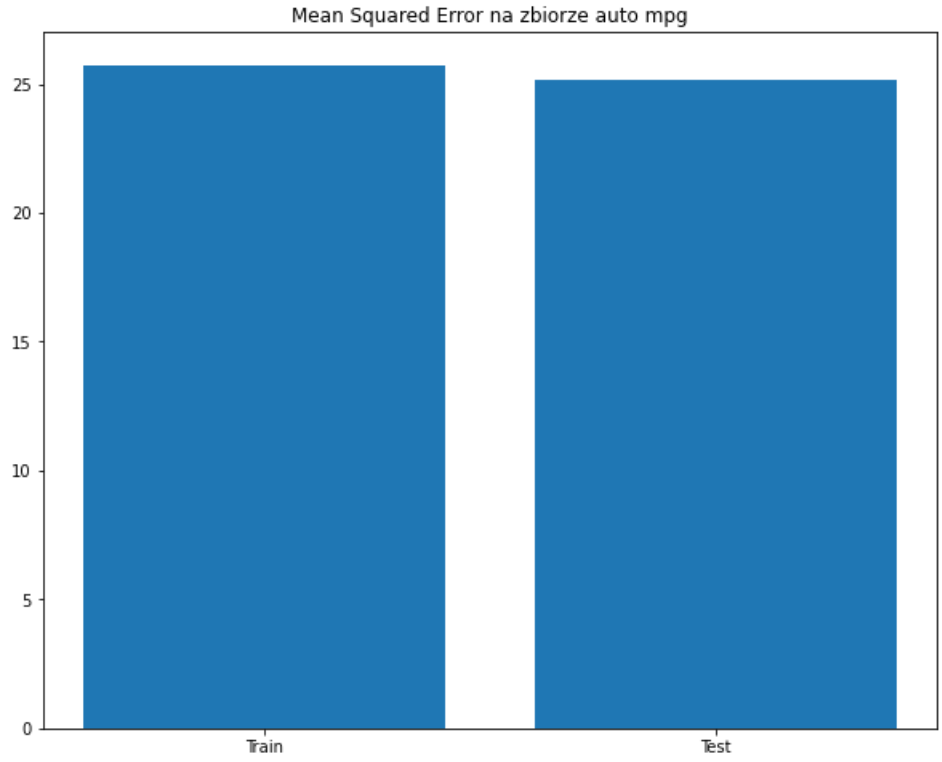
Ustawione parametry algorytmu:

- crossover_ratio = 0.7
- mutation_ratio = 0.2
- crossover = random_layer
- mutation = noise_mutation
- selection_func = elite_selection
- population_size = 100
- input_shape_network=X_mpg_train.shape
- neurons_num=[10,10,10,1]
- activations=[Sigmoid(), Sigmoid(), Sigmoid(), Linear()]
- X = X_mpg_train
- y = y_mpg_train
- loss = Mse()

Po tysiącu iteracji, MSE jakie udało się uzyskać dla najlepszego osobnika w populacji było równe 25.74 na zbiorze treningowym oraz 25.13 na zbiorze testowym.



Rysunek 21: proces ewolucji na zbiorze mpg-auto



Rysunek 22: porównanie MSE na zbiorze mpg-auto

6 Podsumowanie

Algorytmy ewolucyjne mają duży potencjał, z uwagi na bardzo szybkie postępy techniczne i rozwój cyfryzacji. Dostępne są coraz większe zasoby obliczeniowe przez co mają one praktyczne zastosowanie. Algorytmy te bardzo dobrze poradziły sobie w minimalizacji funkcji w AE1 oraz w problemie *cutting stock* w AE2. Jeśli chodzi o ostatnią pracę domową to algorytmy te dawały trochę gorsze wyniki w porównaniu do tradycyjnego uczenia sieci, jednakże dostrajając parametry modelu jeszcze w większym stopniu i przeprowadzając dodatkowe eksperymenty na innych formach mutacji oraz krzyżowania, możnaby poprawić jakość modelu.

Literatura

[1] Strona wykładu. http://pages.mini.pw.edu.pl/karwowskij/metody-inteligencji-obliczeniowej-w-analizie-danych.

[2] Towards data science. https://towardsdatascience.com.