WYDZIAŁ MATEMATYKI I NAUK INFORMACYJNYCH
POLITECHNIKA WARSZAWSKA

Algorytmy genetyczne

SPRAWOZDANIE

Kornel Tłaczała

16 czerwca 2025

Spis treści

AE1: Implementacja algorytmu genetycznego

Cel

Zaimplementowanie algorytmu genetycznego, który będzie w stanie rozwiązać problem optymalizacji funkcji. W tym przypadku celem jest znalezienie minimum dwóch funkcji:

1. $f(x, y, z) = x^2 + y^2 + 2z^2$.
2. funkcja Rastrigina

Implementacja

Model został zaimplementowany w języku Python. Trening przeprowadzany jest po wygenerowaniu populacji początkowej, która losowana jest z rozkładu jednostajnego wokół zera. Następnie, w każdej iteracji, wykonywane są następujące kroki:

1. Selekcja rodziców - wybór osobników do reprodukcji. W tym przypadku zastosowano bardzo prostą selekcję. Wybierany jest pewien procent najlepszych osobników z populacji.
2. Krzyżowanie - tworzenie nowych osobników poprzez krzyżowanie rodziców. W tej implementacji każdy rodzic tworzy dwóch potomków z każdym innym rodzicem. Krzyżowanie polega na wymianie części genotypów rodziców. Wykorzystana została tu metoda `one_point_crossover`, gdzie punkt krzyżowania jest losowany losowo.
3. Mutacja - wprowadzenie losowych zmian w genotypie potomków, aby zwiększyć różnorodność populacji. Prowadzi to do powstania szumu i pozwala na przeszukiwanie całej przestrzeni rozwiązań
4. Finalna selekcja - wybór najlepszych osobników z populacji potomnej. Zastosowano tutaj elityzm - procent najlepszych rodziców zawsze zostaje w populacji. Reszta miejsc jest uzupełniana przez najlepszych osobników.

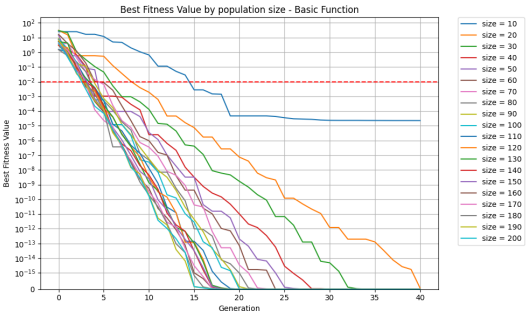
Wyniki

Dla obu funkcji model był w stanie efektywnie znaleźć minimum. Oczywiście w przypadku obu funkcji minimum jest w zerze. Wyniki to zdecydowanie odzwierciedlają:

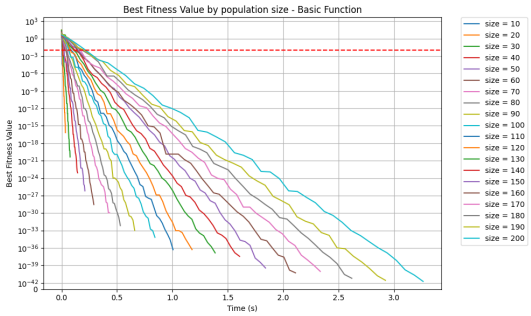
```
================================================================================
================================   basic   =================================
================================================================================
best value: 1.4011307135309994e-63
best vector: [-3.37045296e-32 -1.19666812e-32  7.80813488e-33]

================================================================================
================================   rastrigin   =================================
================================================================================
best value: 0.0
best vector: [-4.32889402e-09  4.98032353e-11  8.32728896e-10 -4.03628216e-10
  1.39484285e-09]
```

Rysunek 1: Wyniki dla obu funkcji

Szybkość osiągania precyzji



(a) Precyzja dla różnych rozmiarów populacji wg. epoki

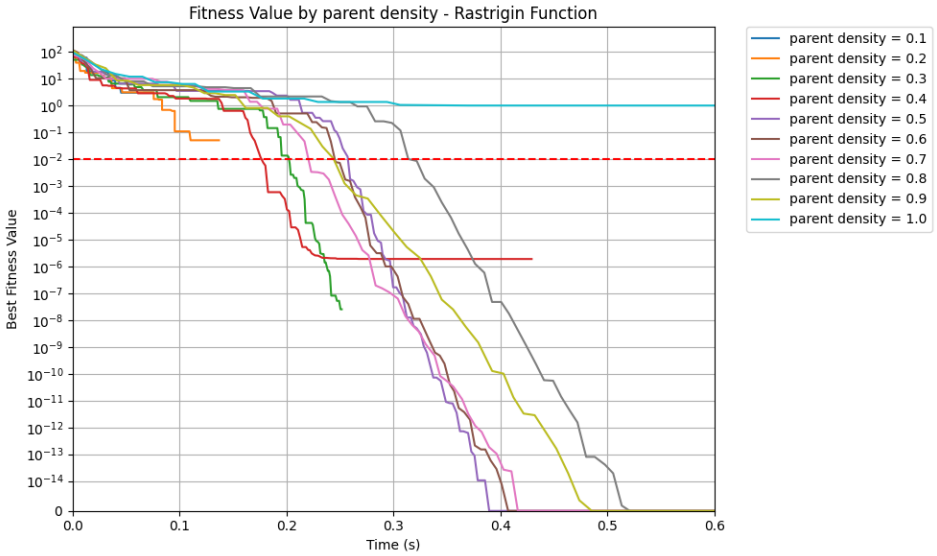


(b) Czas osiągania precyzji dla różnych rozmiarów populacji

Rysunek 2: Zwykła funkcja kwadratowa - opłacalność rozmiaru populacji

Jak widać mniejsze populacje szybciej osiągają większą precyzję. Wynika to z tego, że złożoność algorytmu jest liniowa względem rozmiaru populacji. Jednakże, mniejsze populacje mogą nie być w stanie znaleźć minimum, ponieważ nie mają wystarczającej różnorodności genetycznej. Szczególnie widać to dla funkcji Rastrigina, gdzie mała populacja może łatwo utknąć w lokalnym minimum.

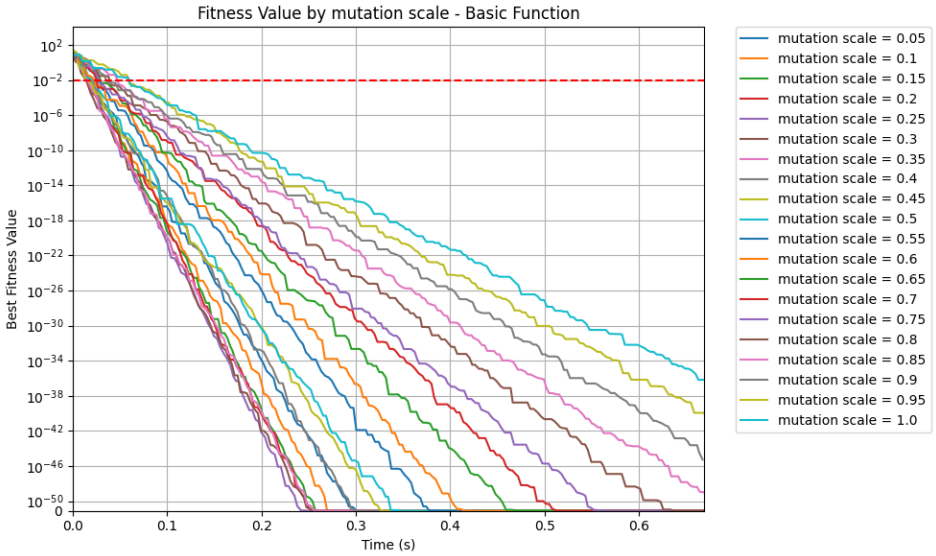
Zależność zbieżności od gęstości selekcji rodziców



Rysunek 3: Zbieżność w zależności od gęstości selekcji rodziców

Jak widać, optymalne wartości gęstości selekcji rodziców są z przedziału $0.5 - 0.7$. Przy zbyt małej gęstości selekcji rodziców, populacja nie osiąga optymalnej precyzji. Zbyt duża gęstosc wydłuża natomiast działanie algorytmu.

Zależność zbieżności od skali mutacji

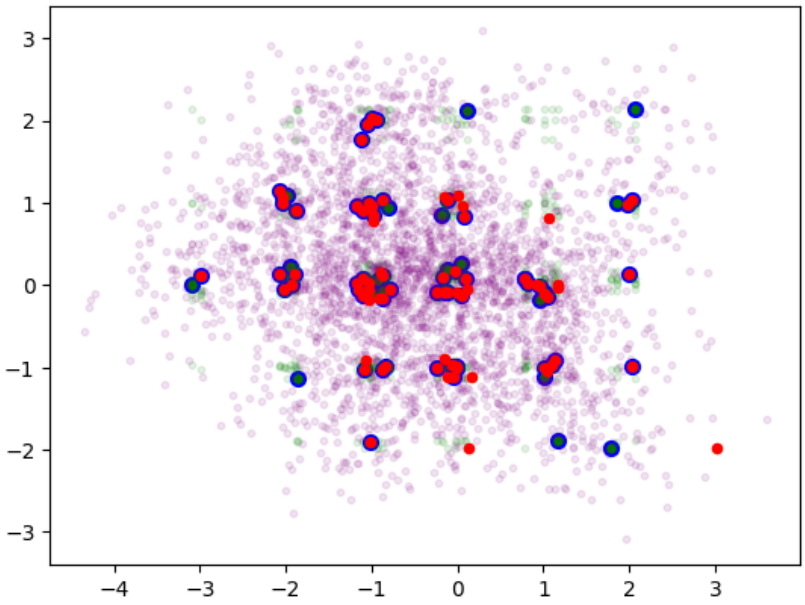


Rysunek 4: Zbieżność w zależności od skali mutacji

Najlepsze skale mutacji są z przedziału $0.05 - 0.4$.

Funkcja Rastrigina

Funkcja Rastrigina jest funkcją o wielu minimach lokalnych, co sprawia, że jest trudna do optymalizacji. Na wykresie czerwone kropki oznaczają nową populację, niebieskie starą populację. Na zielono są dzieci starej populacji a na fioletowo ich mutacje. Dopiero teraz widać, dlaczego zbyt mały rozmiar populacji może zapobiec znalezieniu rozwiązania.



Rysunek 5: Wizualizacja działania algorytmu na funkcji Rastrigina

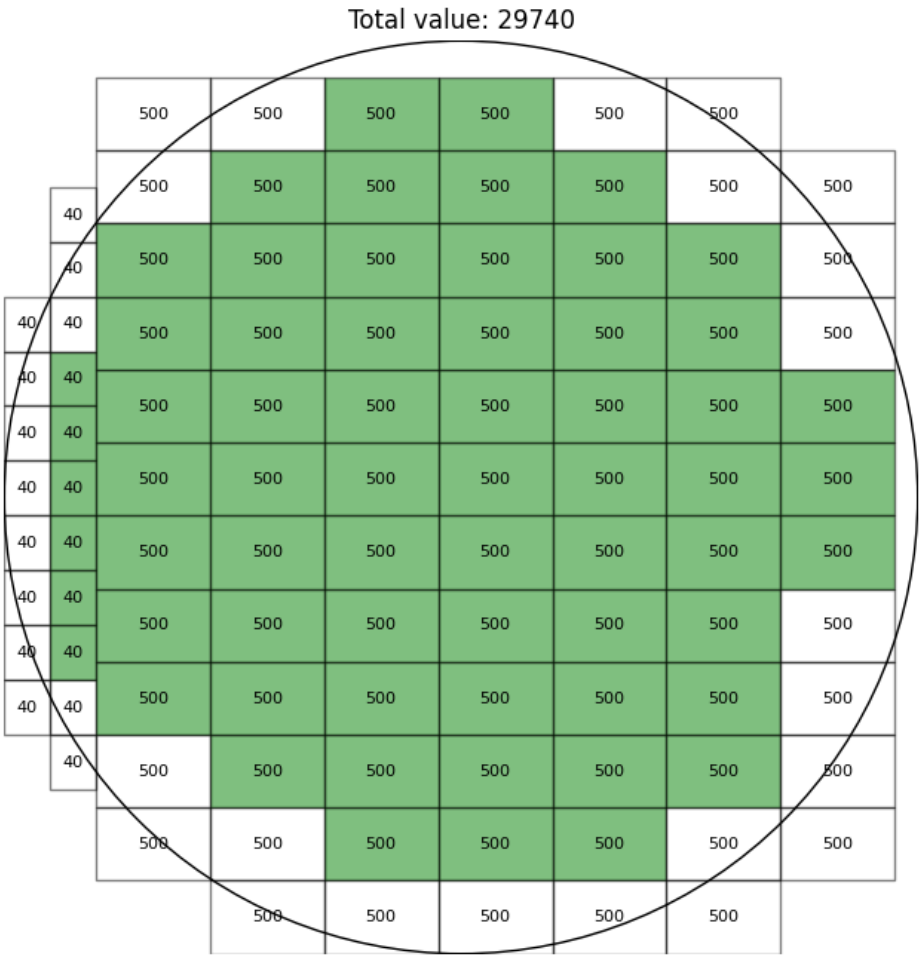
AE2: Wypełnianie koła prostokątami

Cel

Celem zadania jest zaprojektowanie algorytmu, który maksymalizuje powierzchnię prostokątów umieszczonych wewnątrz koła o zadanym promieniu. Algorytm powinien znaleźć taki układ prostokątów, aby suma ich pól była jak największa. Problem ten jest przykładem zadania optymalizacyjnego z ograniczeniami geometrycznymi i może być rozwiązywany m.in. z wykorzystaniem algorytmów ewolucyjnych.

Implementacja

Od razu można zauważyć, że bardzo dużo miejsca w okręgu można stracić poprzez przestrzenie w które nie jest w stanie zmieścić się żaden nowy prostokąt. Warto układać prostokąty tak, aby do siebie przylegały. Najprostszym zatem rozwiązaniem okazało się umieszczenie prostokątów w kolumnach o jednolitym typie prostokąta. Genomem zatem można określić listę rodzajów prostokątów, które będą umieszczane w kolejnych kolumnach. Przykładowe umieszczenie można zaobserwować na poniższym rysunku:



Rysunek 6: Przykładowe umieszczenie prostokątów w kole

Proces treningowy

Trening przebiegał w sposób podobny do treningu algorytmu genetycznego z AE1. Wygenerowana została populacja początkowa, każdy osobnik (układ rodzajów prostokątów w kolumnach) jest oceniany

poprzez obliczenie pola powierzchni prostokątów, które są w stanie zmieścić się w kole. Następnie, w każdej iteracji, wykonywane są następujące kroki:

1. Selekcja rodziców - wybór osobników do reprodukcji. W tym przypadku zastosowano selekcję turniejową. Wybieramy pewien procent (przykładowo 10%) osobników, którzy biorą udział w turnieju. Wybierany jest najlepszy osobnik z tego turnieju. Tak wybieramy rodzica nr. 1 oraz rodzica nr. 2.

2. Krzyżowanie - tworzenie nowych osobników poprzez krzyżowanie rodziców. W tej implementacji każdy rodzic tworzy dwóch potomków z każdym innym rodzicem. Krzyżowanie polega na wymianie części genotypów rodziców. Wykorzystana została tu metoda `one_point_crossover`, gdzie punkt krzyżowania jest losowany losowo. Oczywiście, może się okazać, że kolumny potomka są zbyt szerokie lub zbyt wąskie. W takim przypadku nadmiar kolumn jest ucinany, a brakujące kolumny są uzupełniane losowymi rodzajami prostokątów.

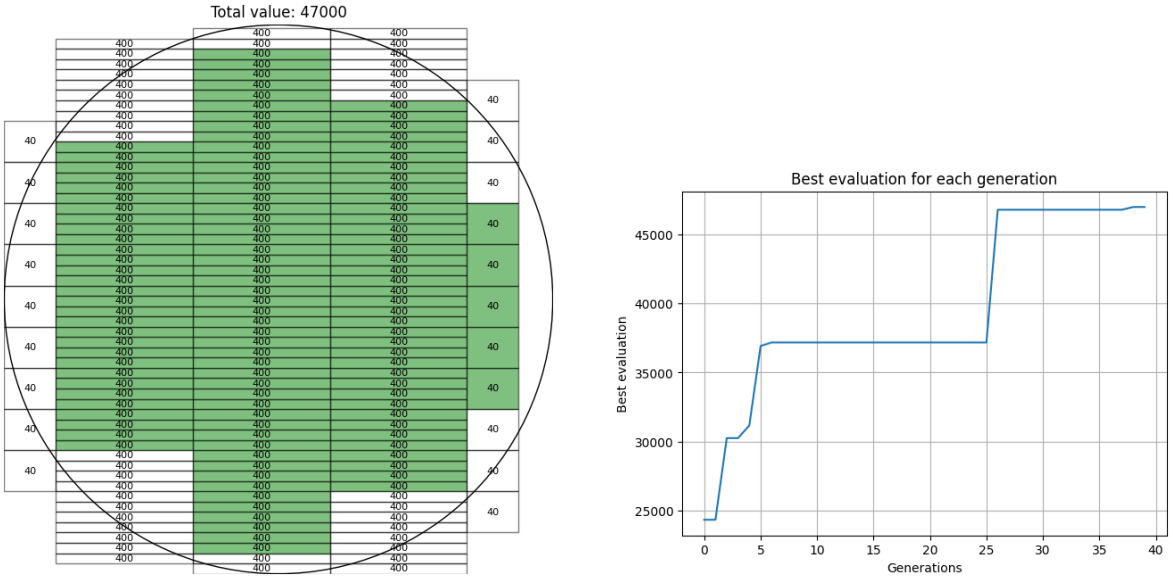
3. Mutacja - wprowadzenie losowych zmian w genotypie potomków, aby zwiększyć różnorodność populacji. Prowadzi to do powstania szumu i pozwala na przeszukiwanie całej przestrzeni rozwiązań. Mutacja polega na losowym przemieszaniu wszystkich kolumn potomka.

4. Finalna selekcja - aby nie tracić jakości modelu najgorszy osobnik nowej populacji jest zastępowany przez najlepszego osobnika z poprzedniej populacji.

Wyniki

Koło o promieniu $r = 800$

Cel: osiągnięcie wartości $Value > 30000$.



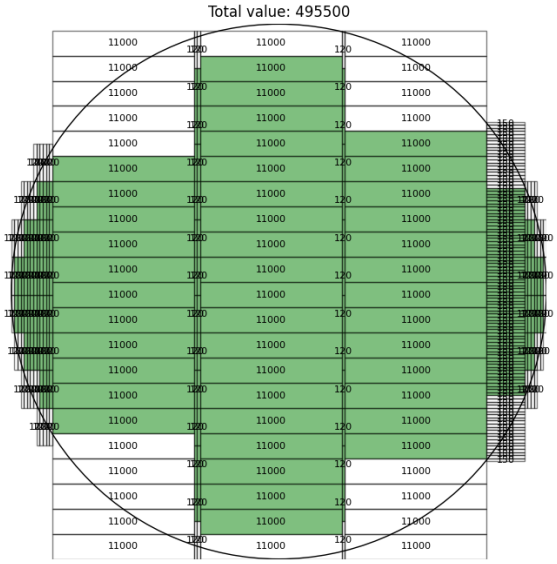
(a) Najlepsze rozmieszczenie prostokątów

(b) Postęp treningu

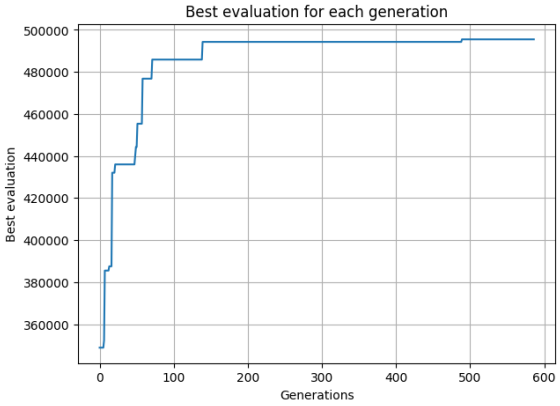
Rysunek 7: Wyniki dla koła o promieniu $r = 800$

Koło o promieniu $r = 850$

Cel: osiągnięcie jak najwyższego wyniku.



(a) Najlepsze rozmieszczenie prostokątów

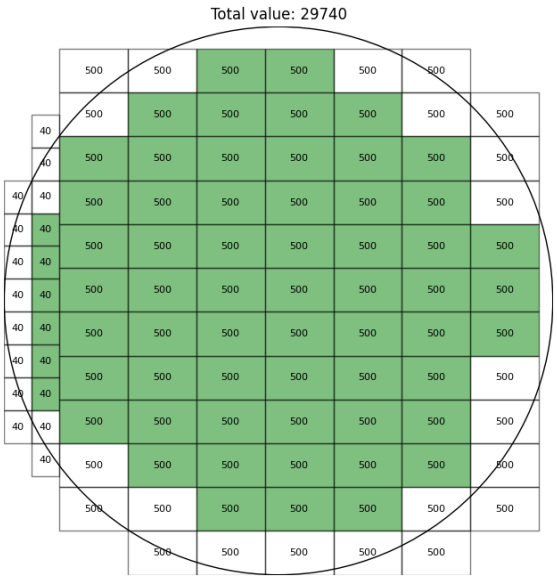


(b) Postęp treningu

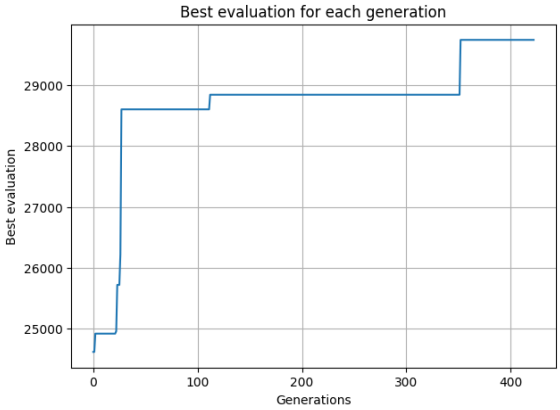
Rysunek 8: Wyniki dla koła o promieniu $r = 850$

Koło o promieniu $r = 1000$

Cel: osiągnięcie wartości $Value > 17500$.



(a) Najlepsze rozmieszczenie prostokątów

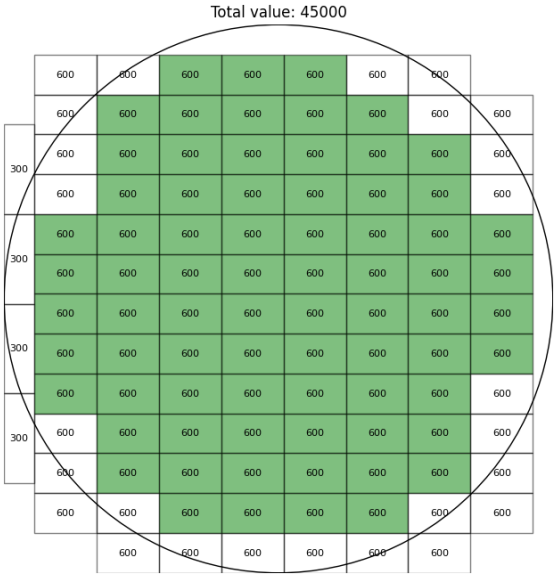


(b) Postęp treningu

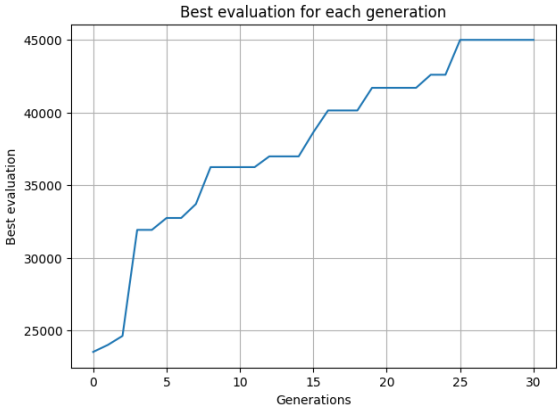
Rysunek 9: Wyniki dla koła o promieniu $r = 1000$

Koło o promieniu $r = 1100$

Cel: osiągnięcie wartości $Value > 25000$.



(a) Najlepsze rozmieszczenie prostokątów

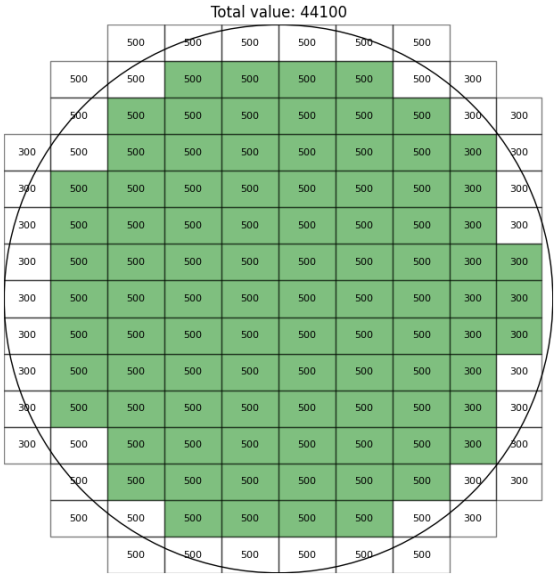


(b) Postęp treningu

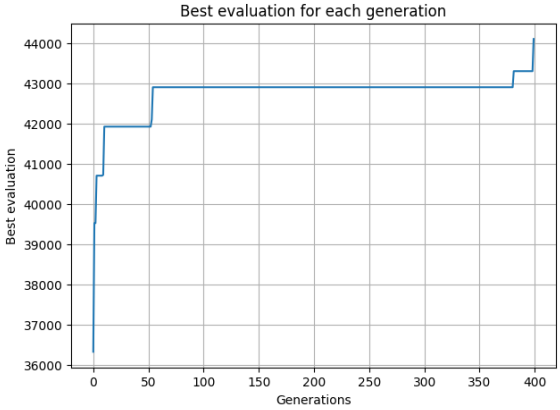
Rysunek 10: Wyniki dla koła o promieniu $r = 1100$

Koło o promieniu $r = 1200$

Cel: osiągnięcie wartości $Value > 30000$.



(a) Najlepsze rozmieszczenie prostokątów



(b) Postęp treningu

Rysunek 11: Wyniki dla koła o promieniu $r = 1200$

AE3: Optymalizacja wag sieci MLP z wykorzystaniem algorytmu genetycznego

Cel

Celem zadania była optymalizacja wag sieci MLP (Multi-Layer Perceptron) z wykorzystaniem algorytmu genetycznego. W tym przypadku celem jest znalezienie wag, które pozwolą na jak najlepsze dopasowanie modelu do danych treningowych. Modele trenowane oraz oceniane były na zbiorach:

1. square-simple (problem regresji jednowymiarowej)
2. multimodal-large (problem regresji jednowymiarowej)
3. iris (problem klasyfikacji)
4. auto_mpg (problem regresji wielowymiarowej)

Implementacja

Do implementacji modelu zosała wykorzystana już istniejąca struktura sieci MLP. Osobnikiem w populacji jest zmodyfikowany obiekt modelu MLP, tak zwany EvoMLP. Wagi oraz biasy sieci są traktowane jako genotyp osobnika. Osobnik jest oceniany przy użyciu natywnej funkcji straty modelu, określanej przy jego inicjalizacji. W przypadku regresji, funkcją oceny jest błąd średniokwadratowy (MSE), a w przypadku klasyfikacji - entropia krzyżowa (cross-entropy). Zadaniem jest zminimalizowanie wartości funkcji straty na zbiorze treningowym dla najlepszego osobnika w populacji.

Proces treningowy

Wygenerowanie populacji początkowej było bardzo naturalnym krokiem, ponieważ domyślnie model MLP jest inicjalizowany z losowymi wagami i biasami. Następnie, w każdej iteracji, wykonywane są następujące kroki:

1. Selekcja rodziców - wybór osobników do reprodukcji. Zastosowano tu selekcję tureniejową, tak jak w AE2. Wybierany jest pewien procent najlepszych osobników z populacji, którzy biorą udział w turnieju. Wybierany jest najlepszy osobnik z tego turnieju. Tak wybieramy rodzica nr. 1 oraz rodzica nr. 2. którzy tworzą parę.

2. Krzyżowanie - tworzenie nowych osobników poprzez krzyżowanie rodziców. W tej implementacji każdy rodzic tworzy dwóch potomków z każdym innym rodzicem. Krzyżowanie polega na pobraniu jednej wagi oraz biasu dla losowego neuronu od drugiego rodzica. Daje to możliwość szybkiej, gwałtownej zmiany predykcji przez osobnika. Zazwyczaj taka zmiana jest niekorzystna, natomiast czasem może ona prowadzić do znacznej poprawy oceny modelu, co jest wypadkową znalezienia lepszego dopasowania do skomplikowanych danych. Szczególnie dobrze widać to na zbiorze multimodal-large.

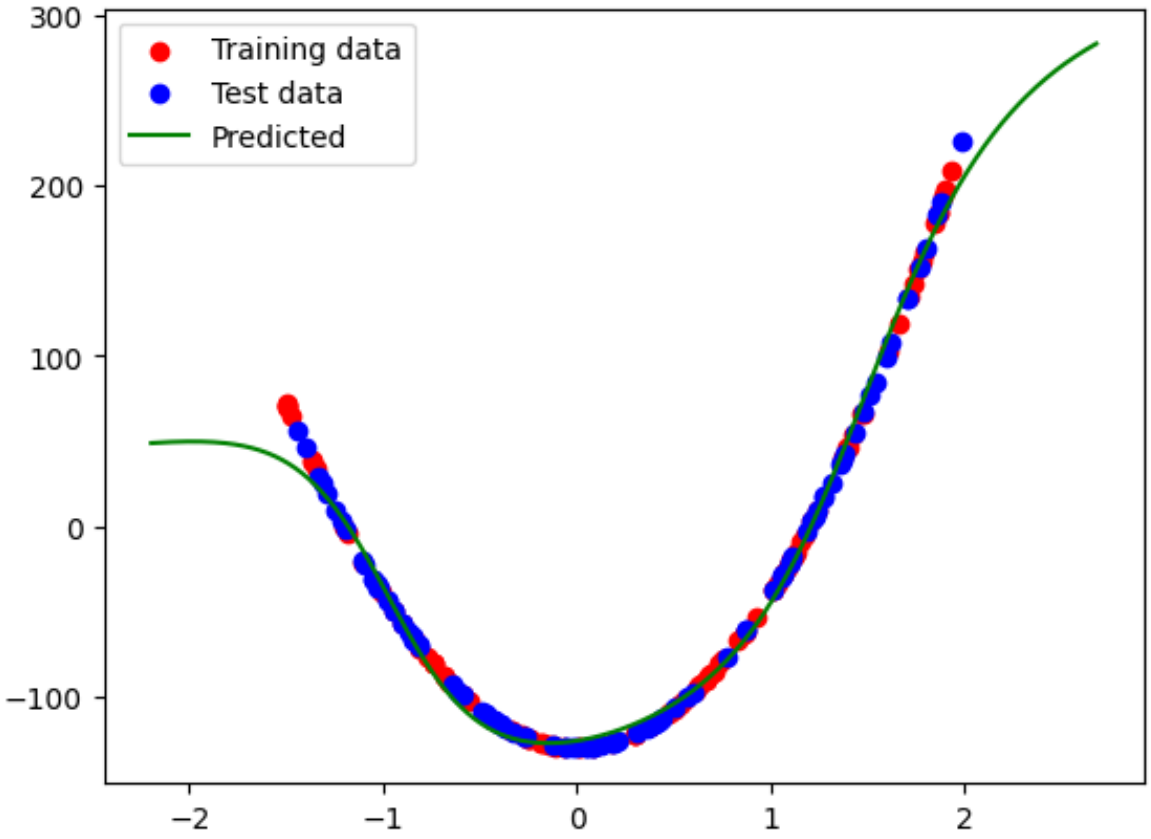
3. Mutacja - tutaj pomysłów było dużo. Pierwsze dwa polegały na zmianie wielu wag naraz przy jednym neuronie, lub zmianie wag na całej warstwie na raz. Okazały się one jednak nieskuteczne, ponieważ prawie nigdy nie powodowały korzystnej mutacji, ewaluacja modelu często stawała w miejscu. Ostatecznie zaimplementowana została mutacja, która dla wybranego neuronu mnoży jego wagę oraz bias przez losową wartość bliską 1. Parametr tej bliskości - `mutation_strength` można dosyć swobodnie dobierać podczas procesu treningowego

4. Finalna selekcja - trafiały tutaj 4 grupy osobników. Wszyscy osobnicy z poprzedniej generacji, wszystkie dzieci, dodatkowo zmutowani rodzice oraz dodatkowo zmutowane dzieci. Bardzo ważne okazało się pozostawianie 10% najlepszych wygenerowanych osobników. Gwarantowało to stabilność modelu i pozwalało na uniknięcie regresji jakościowej. Model okazał się dosyć odporny na spadek różnorodności genetycznej. Duża liczba mutacji w każdej generacji pozwalała na wygenerowanie odmiennych osobników, potencjalnie troszkę lepszych niż rodzice. Ostatecznie, oprócz

10% najlepszych osobników reszta kandydatów była wybierana z rozkładu prawdopodobieństwa proporcjonalnego do ich oceny. Oznacza to, że im lepszy osobnik, tym większa szansa, że zostanie wybrany do kolejnej generacji.

Wyniki

Square-simple



Rysunek 12: Dopasowanie modelu EvoMLP do zbioru square-simple

Dla tego prostego zbioru udało się osiągnąć zbliżone dopasowanie po około 600 epokach. Po 900 epokach wartość funkcji straty wyniosła $MSE = 37.16$. Jest to oczywiście wynik nieporównywalnie gorszy niż ten osiągany przez wsteczną propagację błędu, jednak satysfakcjonujące jest odwzorowanie przez algorytm kształtu funkcji w jej dziedzinie

Iris

Zadanie to jest problemem klasyfikacji 3-klasowej. Dla porównania, wytrenowany został również model MLP przy użyciu wstecznej propagacji błędu. Osiągnął on F1-score na poziomie 0.97, co jest bardzo dobrym wynikiem. Wyraźnie widoczne było jednak, że mimo zastosowania różnych architektur sieci model nie był wstanie zejść poniżej $cross\_entropy = 0.06$

Rysunek 13: Proces treningowy modelu EvoMLP na zbiorze iris

Dla algorytmu ewolucyjnego szczególne korzyści przyniósł trening z większą siłą mutacji. Pozwolił on osiągnąć wyraźnie niższą wartość funkcji straty (0.001). Algorytm osiągnął wynik `F1-score = 1`, co jest lepszym wynikiem niż ten osiągnięty przez MLP z backpropagacją.

Auto MPG

Zaskakująco dobrze model EvoMLP poradził sobie z tym zbiorem danych. Jest to zbiór regresji wielowymiarowej, gdzie celem jest przewidywanie zużycia paliwa w samochodach na podstawie ich parametrów technicznych. Zbiór ten zawiera 7 cech wejściowych i jedną cechę wyjściową (zużycie paliwa).

Zbiór ten wymagał usunięcia kilku wierszy z brakującymi danymi. Dopiero to pozwoliło na wytrenowanie modelu, zarówno MLP jak i Populacji EvoMLP.

Co ciekawe, algorytm ewolucyjny osiągnął lepszy wynik niż klasyczna wsteczna propagacja błędu, nawet przy zastosowaniu zaawansowanych technik takich jak RMSprop, uczenie z momentem, regularyzacja, adaptacyjna zmiana współczynnika nauczania czy zatrzymywanie procesu nauczania w przypadku przeuczania. Mimo tych usprawnień, model trenowany backpropagacją nie był w stanie zejść poniżej $MSE = 6.0$. Algorytm ewolucyjny natomiast ostatecznie osiągnął $MSE < 4.5$, co stanowi zauważalną poprawę jakości predykcji na tym zbiorze danych.

Trening sieci MLP przy pomocy algorytmu ewolucyjnego

In [16]:
```
pop = ModelPopulation(
    # architecture=MLPArchitecture(7, [20, 10, 5], 1),
    architecture=MLPArchitecture(7, [5], 1),
    dataset_name="auto_mpg",
    population_size=100,
)
pop.show_test_loss()
```

```
Best model test loss: 24.28294138898855
```

In [17]:
```
pop.train(epochs=100)
```

```
100%|██████████| 100/100 [00:30<00:00] , Best Loss=9.3049
```

In [18]:
```
pop.train(epochs=100)
```

```
100%|██████████| 100/100 [00:29<00:00] , Best Loss=6.5869
```

In [19]:
```
pop.train(epochs=100)
```

```
100%|██████████| 100/100 [00:29<00:00] , Best Loss=5.6827
```

In [20]:
```
pop.train(epochs=500)
```

```
100%|██████████| 500/500 [02:29<00:00] , Best Loss=4.6162
```

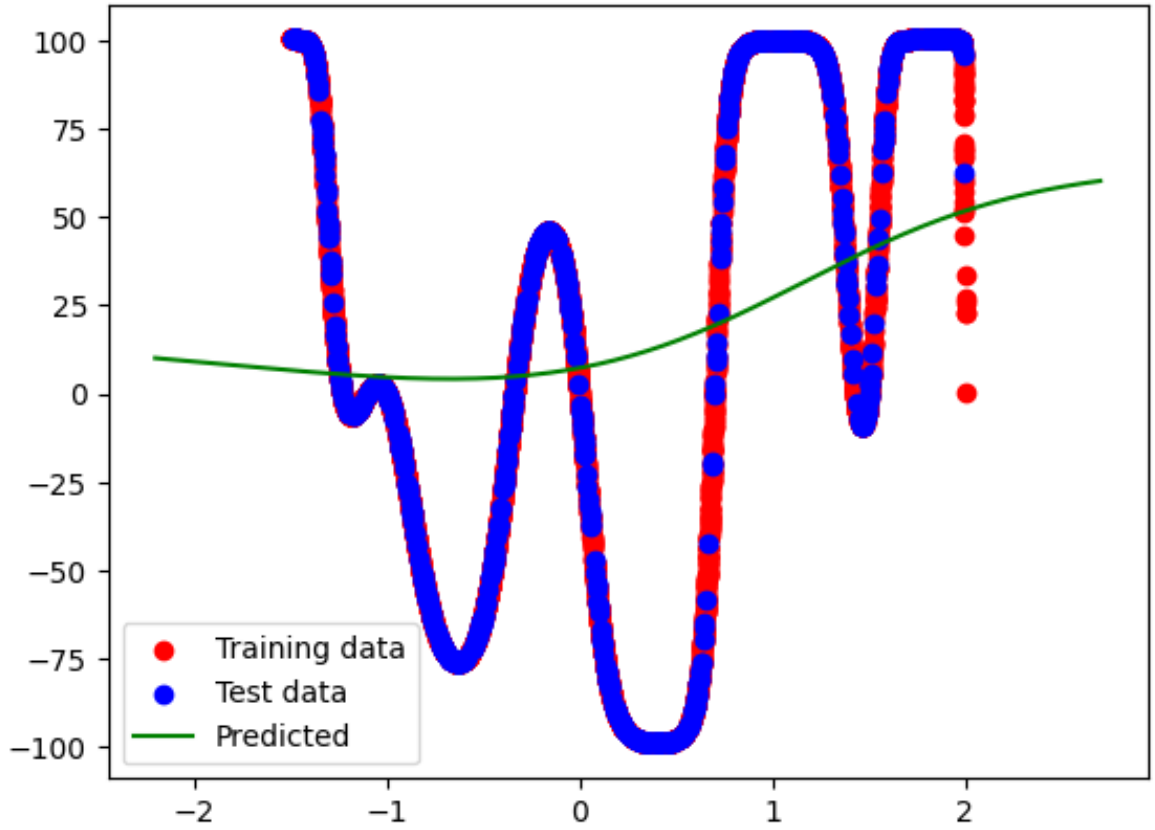
In [22]:
```
pop.train(epochs=500, mutation_strength=0.01)
```

```
100%|██████████| 500/500 [02:25<00:00] , Best Loss=4.4578
```

Rysunek 14: Proces treningowy modelu EvoMLP na zbiorze auto mpg

Multimodal-large

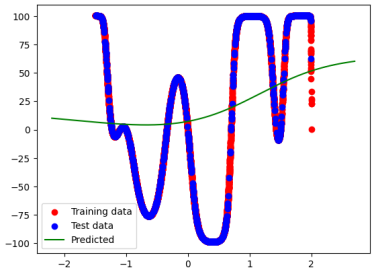
Największym zaskoczeniem podczas testowania algorytmu ewolucyjnego była jego zdolność do radzenia sobie z bardzo skomplikowanym zbiorem multimodal-large. Jest to zbiór regresji jednowymiarowej, gdzie celem jest przewidywanie wartości funkcji o wielu minimach lokalnych.



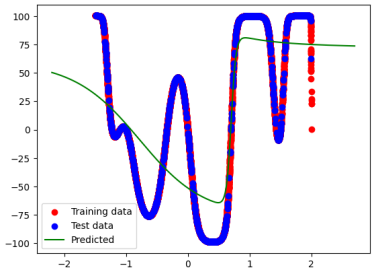
Rysunek 15: Początkowe dopasowanie populacji EvoMLP do zbioru multimodal-large

Jak widać, zadaniem jest tutaj predykcja wartości skomplikowanej funkcji, wymagającej agresywnych zmian wag podczas procesu treningowego. Dużo w procesie treningowym pomagało regularne dostosowywanie siły mutacji. Okresowe zwiększanie siły mutacji pozwalało na znalezienie lepszego rozwiązania, wyraźnie odmiennego od obecnych w populacji, co pozwoliło dopasować się do nieregularnego kształtu funkcji. Następnie zmniejszenie siły mutacji pozwalało na bliższe dopasowanie znalezionego rozwiązania i przyśpieszało postęp.

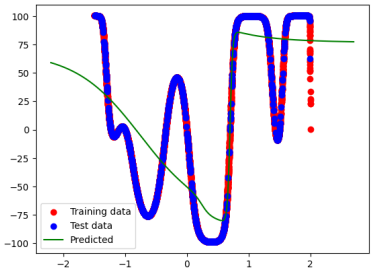
Historia procesu treningowego modelu EvoMLP do zbioru multimodal-large



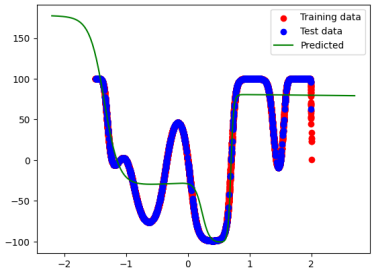
(a) Epoka 0



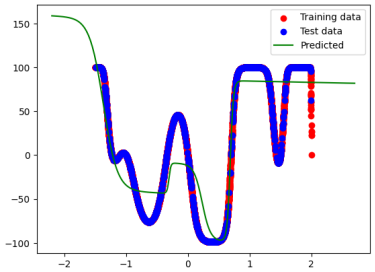
(b) Epoka 200



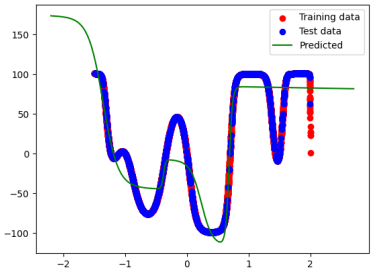
(c) Epoka 300



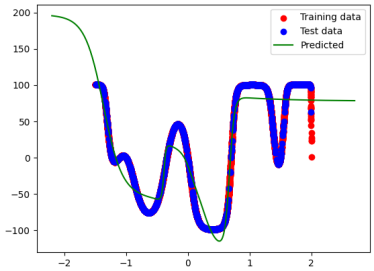
(a) Epoka 800



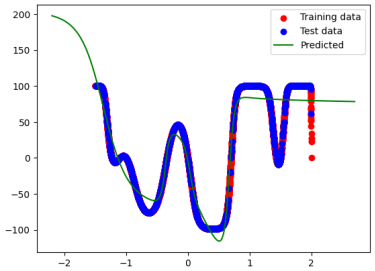
(b) Epoka 1200



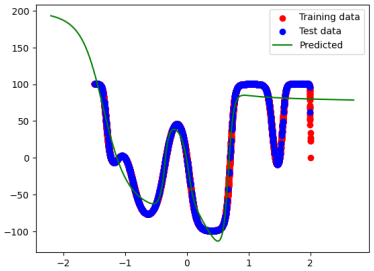
(c) Epoka 1300



(a) Epoka 1900



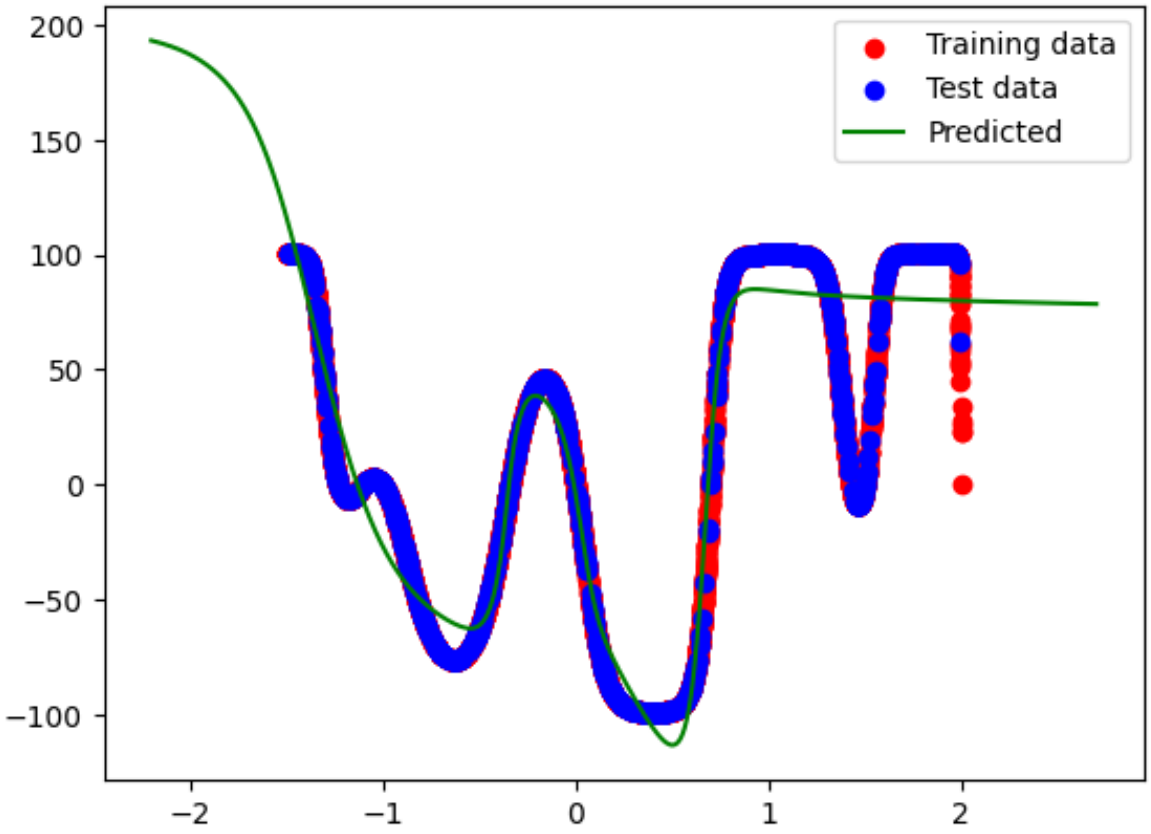
(b) Epoka 2900



(c) Epoka 3300

Zdecydowaną wadą tego algorytmu jest jego czas działania. Lepszą ostateczną skuteczność zapewniają większe populacje, które z kolei wolniej przeszukują przestrzeń rozwiązań, gdyż generują dużo podobnych osobników.

Ostateczne dopasowanie na zbiorze multimodal-large



Rysunek 19: Ostateczne dopasowanie modelu EvoMLP do zbioru multimodal-large