Wydział Matematyki i Nauk Informacyjnych
Politechnika Warszawska

Metody Inteligencji Obliczeniowej w Analizie Danych
Algorytmy Ewolucyjne

Raport z laboratorium

Autor:
Konrad Komisarczyk

Prowadzący:
Maciej Żelaszczyk

Warszawa, 2022

Spis treści

1. Wstęp

1.1. Cel laboratorium

Tematem laboratorium były algorytmy ewolucyjne. Celem laboratorium było zaimplementowanie 3 algorytmów wykorzystujących tą metaheurystyke rozwiązujących 3 różne problemy.

Najpierw rozwiązywaliśmy podstawowy problem optymalizacyjny - optymalizację funkcji w $\mathbb{R}^n$. Potem zaproponowaliśmy własne rozwiązanie wariantu Cutting Stock Problem polegającego na zmieszczeniu w kole prostokątów o maksymalnej sumarycznej wartości. Na końcu przetestowaliśmy zastosowanie algorytmów ewolucyjnych do uczenia sieci neuronowych. Przydała się tu implementacja sieci neuronowej ze wcześniejszego laboratorium.

W każdym zadaniu implementowaliśmy jakiś algorytm, a nastepnie testowalismy go na rżnych przykładach. Starałem się w każdym przypadku przeprowadzić jakieś ciekawe badania, których wyniki mogłyby dać mi jakieś wnioski dotyczące hiperparametrów algorytmów lub samej ich natury.

Rozwiązując kolejne zadania można było zauważyć, że wszystkie implementacje mają podobną strukturę narzucaną przez metaheurystykę. Najważniejszą częścią projektowania algorytmu ewolucyjnego jest wybór reprezentacji rozwiązania oraz operatorów mutacji i krzyżowania. Przyjmowało to różne formy w zależności od problemu. Istotny jest także sposó selekcji. Dodatkowo rozwiązanie rozszerzać można o inne kroki bazujące na wiedzy dziedzinowej, co na przykład zrobiłem w zadaniu 2.

1.2. Kod i reprodukcja

Implementacja każdego zadania oraz kod eksperymentów, a w szczególności kod potrzebny do zreprodukowania wszystkich wykresów znajdujących się w raporcie dostępne są w repozytorium Github pod adresem https://github.com/konrad-komisarczyk/EvolutionaryAlgorithms.

Rozwiązanie pierwszego zadania znajduje się w folderze `Basic`.

Rozwiązanie drugiego zadania znajduje się w folderze `StockCutting`.

Rozwiązanie trzeciego zadania znajduje się w folderze `NeuroEvolution`.

2. AE1 - Optymalizacja funkcji kwadratowej i funkcji Rastrigina.

2.1. Opis zadania

Celem zadania była implementacja prostego algorytmu ewolucyjnego do optymalizowania funkcji w przestrzeni euklidesowej.

Sposób przeprowadzenia mutacji i krzyżowania były zadane z góry: mutacja gaussowska i krzyżowanie jednopunktowe. Mutacja gaussowska jest przesunięciem punktu w każdym wymiarze o odległość pochodzącą z rozkładu normalnego o określonej wariancji. Krzyżowanie jednopunktowe tworzy potomka jako wektor będący połączeniem prefiksu wektora jednego z rodziców i sufiksu wektora drugiego rodzica.

Zaimplementowany algorytm należało przetestować w minimalizacji dwóch funkcji:

— funkcji kwadratowej w $\mathbb{R}^3$: $f(x) = x^2 + y^2 = 2z^2$
— funkcji Rastrigina w $\mathbb{R}^5$: $f(x) = 50 + \sum_{i=1}^{n}[x_i^2 - 10cos(2\pi x_i)]$

Obie funkcje w zerze przyjmują wartość minimalną równą 0.

2.2. Opis zaimplementowanego algorytmu

Rozwiązanie problemu zaimplementowałem w języku Python. Rozwiązaniem jest klasa `Population` rozwiązująca algorytmem ewolucyjnym minimalizację dowolnej funkcji w $\mathbb{R}^n$. Klasa udostępnia 3 istotne dla nas metody:

— Konstruktor z argumentami:
  — `eval_f` - funkcja celu, przyjmująca obiekt sprowadzalny do wektora numpy'owego, zwracająca porównywalną wartość
  — `size` - liczba osobników w populacji
  — `min_value`, `max_value` - dolne i górne ograniczenia (po współrzędnych) obszaru, z którego losowana będzie populacja początkowa. Wektory w $\mathbb{R}^n$, obiekty sprowadzalne do wektora numpy'owego.

  Konstruktor inicjalizuje populację początkową losując punkty z rozkładu jednostajnego we wskazanym n-wymiarowym prostokącie
— `iteration` - funkcja wykonująca iterację algorytmu, przyjmuje parametry:
  — `n_mutations` - liczba mutacji
  — `sigma` - wariancja przesunięcia w mutacji
  — `n_crossings` - liczba krzyżowań
  — `elite_count` - liczność elity
  — `tournament_size` - rozmiar turnieju

  W iteracji najpierw generowane są osobniki powstałe przez mutacje losowych osobników. Następnie osobniki powstałe przez krzyżowanie losowych par osobników (możliwe, aczkolwiek mało prawdopodobne, krzyżowanie osobnika z sobą samym). Populacja rozszerzana jest o osobniki z mutacji i krzyżowania. Następnie przeprowadzana jest selekcja: wybierane jest `elite_count` osobników o najniższej wartości funkcji ewaluacji, a następnie do osiągnięcia

oczekiwanego rozmiaru populacji wyjściowej osobniki wybierane są jako zwycięzcy turniejów. Turniej to `tournament_size` losowo wybranych osobników z dostępnej puli, zwycięzcą jest osobnik o najniższej funkcji ewaluacji.

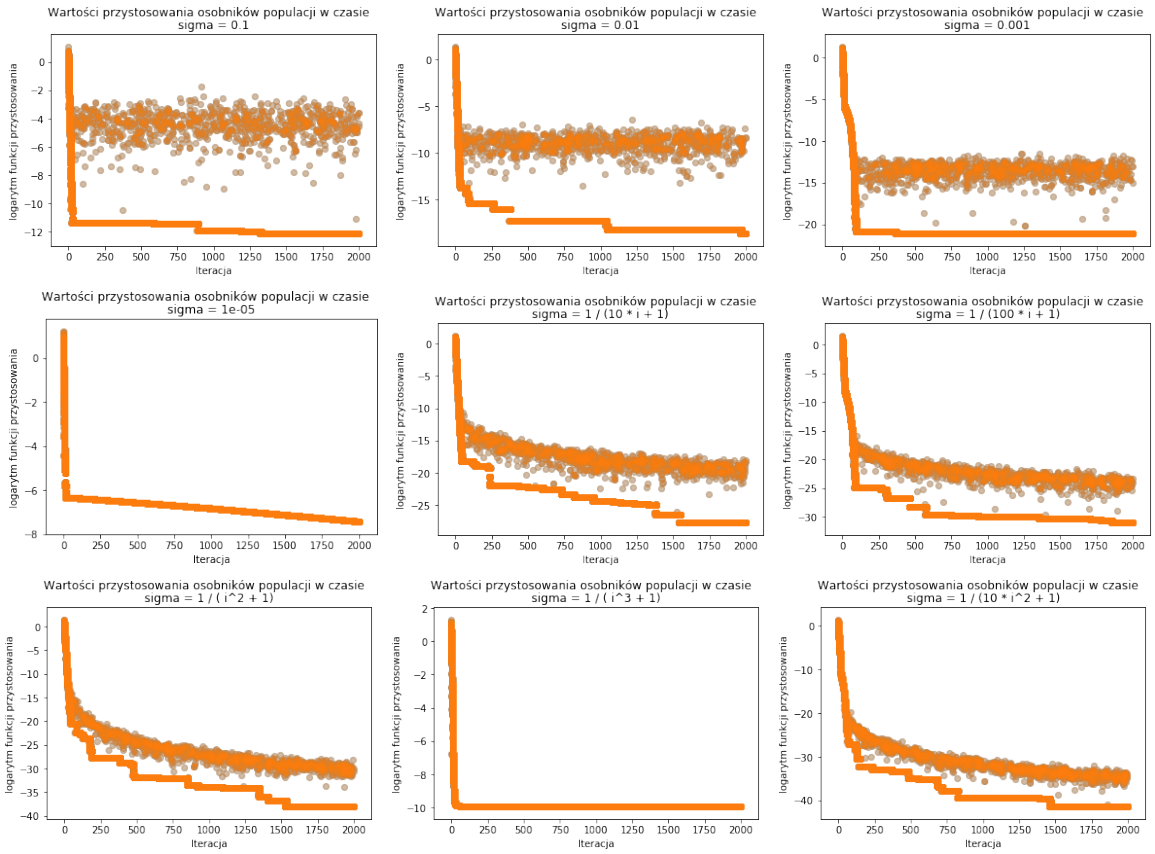
— `select_best` - zwraca osobnika o najlepszej funkcji przystosowania w aktualnej populacji. n-wymiarowy wektor numpy'owy.

Implementacja algorytmu znajduje się w pliku `Basic.py`, natomiast w pliku `Basic.ipynb` znajduje się rozwiązanie zadania dla

2.3. Minimalizacja funkcji kwadratowej

W ramach testowania algorytmu przeprowadziłem 2 eksperymenty.

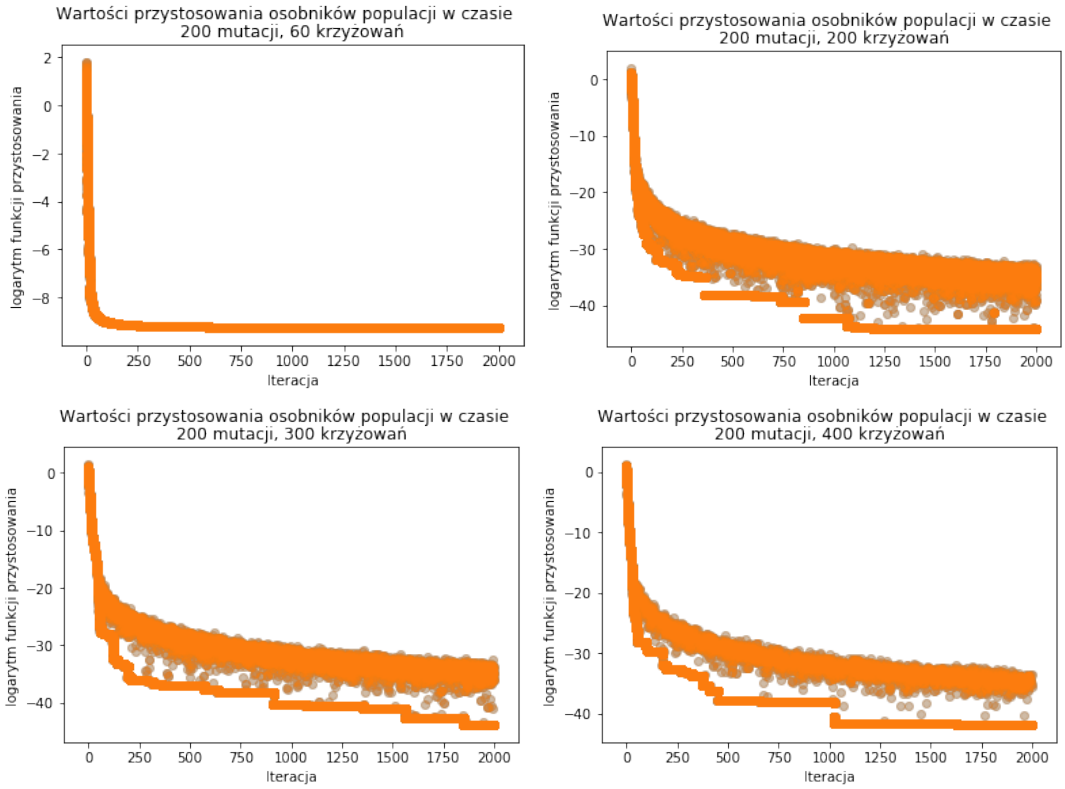
W pierwszym przyjąłem stały rozmiar populacji (200 osobników), stałą liczbę mutacji (60 = 30% populacji), stałą liczbę krzyżowań (140 = 70% populacji), stały rozmiar elity (32) i stały rozmiar turnieju (4). Porównałem natomiast różne wartości `sigma` - wariancji mutacji. Zbadałem wartość funkcji przystosowania wszystkich osobników w czasie. Poniżej znajdują się wykresy pokazujące, jak zmieniał się logarytm funkcji przystosowania w czasie dla różnych wariancji.



Rys. 2.1. Logarytm wartości przystosowania osobników w czasie. Porównałem zarówno wariancje stałe w czasie jak i zależne od numeru iteracji i.

Zmniejszanie wariancji mutacji do pewnego stopnia poprawia dokładność algorytmu kosztem szybkości zbieżności. Zbyt duże jej zmniejszenie może prowadzić do zbyt wolnej zbieżności, jak w przypadku `sigma =` 10^{-5}.

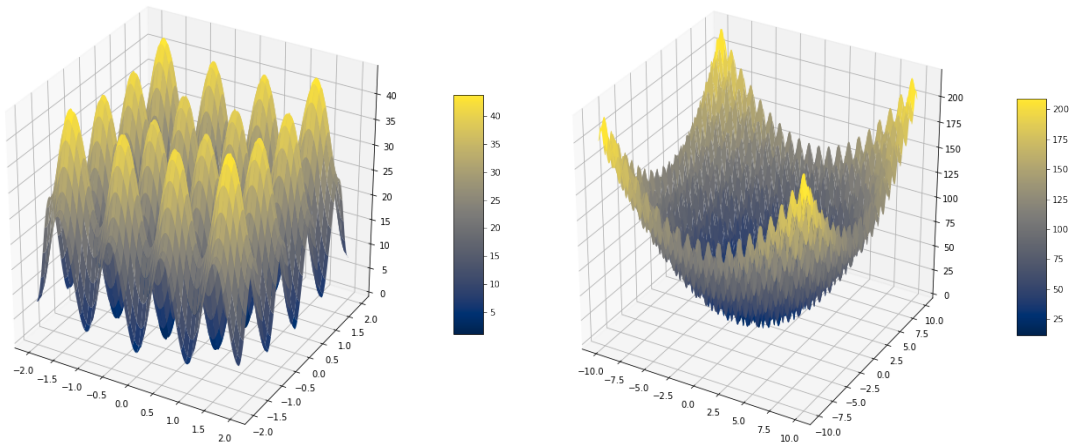
Do drugiego i trzeciego eksperymentu wybrałem najlepiej sprawującą się w poprzednim eksperymencie wariancję `sigma` $= \frac{1}{10i^2+1}$. Porównałem, jaki wpływ ma liczba mutacji na szybkość zbieżności. Poniżej umieszczam wykresy pokazujące wyniki tych eksperymentów:



Rys. 2.2. Logarytm wartości przystosowania osobników w czasie. Porównanie liczby krzyżowań przy stałych pozostałych argumentach.

Widzimy, że zwiększanie liczby mutacji może poprawić zbieżność do pewnego momentu, w tym celu powiększanie jej powyżej pewnego poziomu nie ma już sensu.

2.4. Minimalizacja funkcji Rastrigina



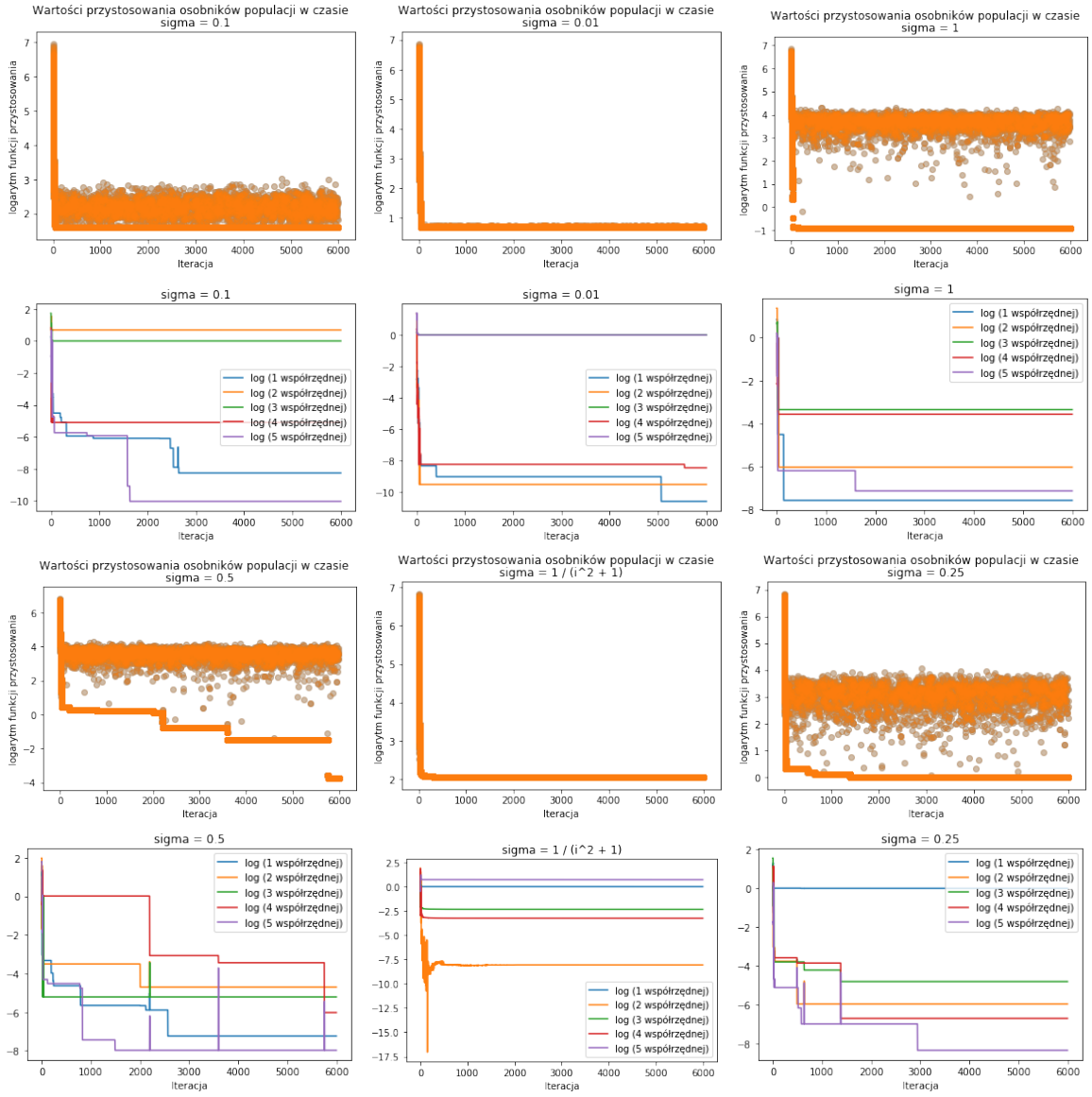
Rys. 2.3. Dwuwymiarowa funkcja rastrigina.

Funkcja Rastrigina ma wiele minimów lokalnych. Możemy zatem spodziewać się, że algorytm będzie utykał w takich minimach. Musimy wystartować tutaj algorytm w większym przedziale

niż $(-1, 1)$, bo funkcja Rastrigina ma w tym przedziele mało mimów lokalnych, chcemy narazić nasz algorytm na utykanie w minimach lokalnych nie będących minimem lokalnym. Wystartujemy w $(-20, 20)$.

Tutaj także porównam zbieżność w zależności od wariancji mutacji. Wszystkie pozostałe parametry przyjmuję takie same, jak przy porównaniu dla funkcji kwadratowej.

Jako, że jest tutaj ryzyko utykania w minimum lokalnym, w którym wartość jest bliska 0, to zwizualizuję tu także pozycję najlepszego osobnika jako logarytmy wartości jego współrzędnych. Jeżeli funkcja trafia we właściwe minimum, to wartości te powinny zbiegać do 0, w przeciwnym wypadku mogą być duże (> 0.5).



Rys. 2.4. Porównanie wariancji mutacji przy funkcji Rastrigina.

Tutaj już nie udało się osiągnąć tak dobrych wyników. Najlepszy wynik udało się osiągnąć w eksperymencie z wariancją równą 0.5. Z wykresu pokazującego wartości lidera populacji po współrzędnej można zobaczyć, że rzeczywiście jest to rozwiązanie znajdujące się blisko minimum globalnego. Jednak z jakiś powodów nie udało mu się zbiegnąć w okolicy tego minimum bliżej zera, mimo, że widzimy, że najlepsze rozwiązanie "wpadło w lej" wokół zera już wiele iteracji wczesniej.

3. Wypełnianie koła prostokątami

3.1. Opis zadania

Zadanie polegało na sformułowaniu, zaimplementowaniu oraz przetestowaniu algorytmu ewolucyjnego rozwiązującego wariant Stock Cutting Problem.

Wariant problemu polegał na znalezieniu takiego wypełnienia koła o danym promieniu niepokrywającymi się prostokątami, aby suma wartości prostokątów była jak największa. Dany jest zbiór rodzajów prostokątów opisanych rozmiarami oraz wartością prostokąta.

Do zadania dołączone było 5 zbiorów rodzajów prostokątów oraz promienie kół dla tych zbiorów. Należało przekroczyć dane sumaryczne wartości prostokątów dla każdego zbioru.

3.2. Rozwiązanie

Osobnikiem jest zbiór prostokątów w kole. Opisany jest jako lista prostokątów opisanych wymiarami, wartością oraz pozycją w układzie współrzędnych. Każdy osobnik jest poprawnym rozwiązaniem, tzn prostokąty nie nachodzą na siebie i nie wychodzą poza koło. Zakładam, że wszystkie prostokąty są równoległe do osi układu współrzędnych. Nie pokrywam w ten sposób całej przestrzeni możliwych rozwiązań. Jednak założenie takie bardzo upraszcza kodowanie i obliczenia, a możemy podejrzewać, że można w ten sposób osiągnąć rozwiązania o bardzo dobrej wartości funkcji przystosowania, bliskiej optymalnej i jest to naturalnym pomysłem na próbę rozwiązania tego problemu.

Mutacja polega na wybraniu losowego rodzaju prostokąta, a następnie wybraniu losowych jego współrzędnych w kole. Następnie prostokąt ten jest dodawany do rozwiązania, a wszystkie obecne już prostokąty, z którymi on koliduje są usuwane.

Krzyżowanie polega na wybraniu losowej cięciwy w kole. Potomek zachowuje wszystkie prostokąty ojca znajdujące się nad cięciwą lub na niej oraz te prostokąty matki, które nie kolidują z zachowanymi prostokątmi ojca. W uproszczeniu możnaby powiedzieć, że znad cięciwy bierze prostokąty od ojca, a spod cięciwy od matki.

Bardzo istotnym elementem rozwiązania jest algorytm **poprawiania** rozwiązań. Składa się on z 4 etapów: zamiatanie w lewo, rozrost w prawo, zamiatanie w dół, rozrost w górę.

Zamiatanie polega na ściągnięciu grawitacją wszystkich prostokątów w ustalonym kierunku. Znajdywane jest takie przesunięcie prostokątów w tym kierunku, że zakładając, że przesunięcie odbywałoby się z pewną stałą prędkością, to żaden prostokąt nie przekroczy w trakcie przesuwania granic innego prostokąta, ani granic koła, przy czym przesunięcie każdego prostokąta będzie maksymalne.

Rozrost polega na dodawaniu do prostokątów w określonym kierunku prostokątów losowo wybranego rodzaju, póki jest to możliwe. Prostokąty dodawane mają wspólny wierzchołek z prostokątem do którego są dodawane. W przypadku rozrostu w prawo jest to prawy dolny wierzchołek prostokąta, od którego rozrastamy i lewy dolny wierzchołek dodawanego prostokąta. Analogicznie w przypadku rozrostu w górę jest to lewy górny wierzchołek prostokąta, od którego rozrastamy i lewy dolny wierzchołek dodawanego prostokąta.

Rozwiązania powstałe w wyniku mutacji lub krzyżowania są **każdorazowo poprawiane**.

W każdej iteracji wykonywana jest określona liczba krzyżowań i mutacji. Kandydaci do nich wybierani są losowo.

Selekcja przeprowadzana jest w dwóch etapach. W pierwszym wybierana jest **elita** o określonej liczości - osobniki o największej funkcji przystosowania (sumie wartości prostokątów). Następnie przeprowadzana jest selekcja **ruletkowa**. Każdy osobnik z pozostałych może zostać wybrany do następnego pokolenia z prawdopodobieństwem proporcjonalnym do jego przystosowania.

Osobniki inicjalizowane są poprzez dodawanie do koła określonej liczby losowych prostokątów w losowych miejscach koła, a następnie poprawienie tych rozwiązań.

Operacje poprawiania rozwiązań są dosyć złożone obliczeniowo. Jednak myślę, że są one zdecydowanie warte wykorzystania. Być może możnaby rozważyć nie wykonywanie ich na każdym osobniku, aby przyspieszyć działanie algorytmu.

Selekcja ruletkowa przeprowadzona w ten sposób jest bardzo zbliżona do wyboru losowego, ponieważ wartości przystosowania są bardzo duże i niewiele różnią się od siebie dla różnych osobników. Jednak za zachowywanie najlepszych rozwiązań odpowiada elitaryzm, a z pozostałych rozwiązań chcemy pozwolić na zachowywanie słabszych, aby pozwolić na większe modyfikacje rozwiązań.

Wartym rozważenia usprawnieniem, które możnaby wprowadzić jest usprawnienie losowania rodzajów dodawanych prostokątów. Prostokąty losujemy przy mutacjach i inicjalizacji osobników. Rodzaje wybierane są z równym prawdopodobieństwem. Być może prawdopodobieństwo proporcjonalne do gęstości prostokąta (stosunku wartości do powierzchni) mogłoby dać lepsze wyniki.

3.3. Implementacja

Algorytm zaimplementowałem w języku Python. Do rozwiązywania problemu służy klasa `Evolution` reprezentujaca populację. Klasa udostępnia następujące metody:

— Konstruktor inicjalizujący populację o danej liczbie osobników, w kole o danym wymiarze, dla danego zbioru danych spełniającego taki interfejs, jak zbiory dołączone do zadania.
— `iter` - funkcja wykonująca na populacji pojedynczą iterację. Ma następujące argumenty:
  — `n_mutations` - liczba mutacji
  — `n_crossings` - liczba krzyżowań
  — `elite_count` - liczność elity
— `best` - zwraca najlepszego osobnika populacji

Poza tym można przeglądać wszystkich osobników populacji znajdujących się w polu `population`.

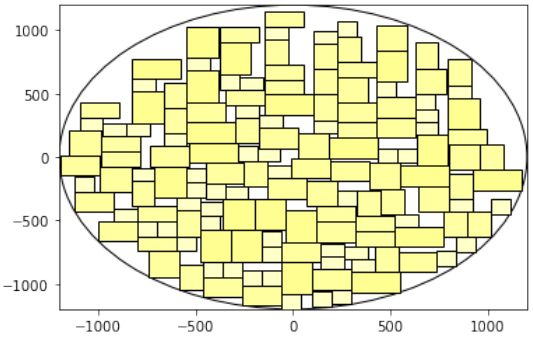
Klasa osobnika populacji to zbiór prostokątów wewnątrz koła. Można go wyrysować za pomocą metody `plot` oraz policzyć ewaluacje za pomocą `evaluate`. Na wykresie przedstawiającym osobnika prostokąty pokolorowane są różnymi kolorami - od białego, przez żółty, potem czerwony - im kolor ciemniejszy, tym stosunek wartości prostokąta do jego powierzchni jest większy.

3.4. Rozwiązania dla zadanych zbiorów

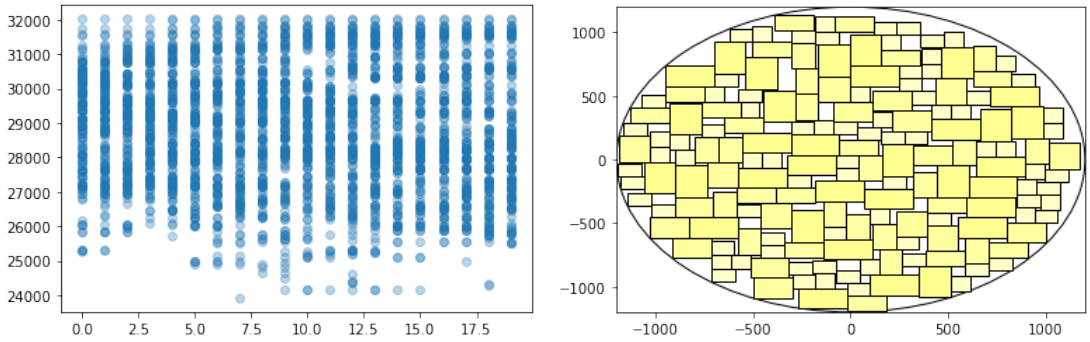
3.4.1. r = 1200

Przyjąłem rozmiar populacji równy 160. Liczność elity wybrałem równą 20. Przyjąłem liczbe mutacji oraz krzyżowań równą 40 w każdej iteracji.

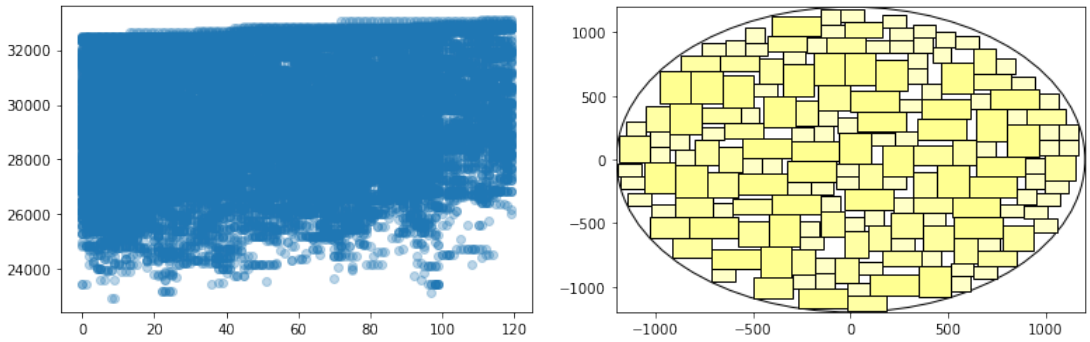
Poniżej przedstawię jeden przebieg algorytmu na tym zbiorze dla tych hiperparametrów:



Rys. 3.1. Najlepsze wypełnienie koła w początkowej populacji. Wartość przystosowania wynosi 30140.



Rys. 3.2. Najlepsze wypełnienie koła po 20 iteracjach oraz wykres wartości przystosowania osobników w czasie (od liczby iteracji). Wartość przystosowania wynosi 32020.



Rys. 3.3. Najlepsze wypełnienie koła po 120 iteracjach oraz wykres wartości przystosowania osobników w czasie (od liczby iteracji). Wartość przystosowania wynosi 33100.

Widzimy, że wzrost wartości przystosowania jest wolny. Widać natomiast, że wykonywanie większej liczby iteracji zapewne pozwoliłoby na dalszy jej wzrost.

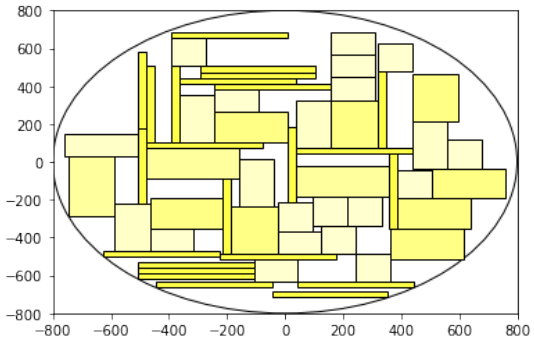
Minimalną wartością przystosowania, jaką musiliśmy osiągnąć dla tego zadania było 30000. Już bez wykonywania żadnych iteracji udało się przebić ten wynik. To pokazuje, jak silny jest algorytm poprawiania rozwiązań.

Wolny wzrost funkcji przystosowania sugeruje, że warto przetestować większą liczbę krzyżowań lub mutacji i wykonać więcej iteracji. Postaram się to zrobić w następnych zbiorach.

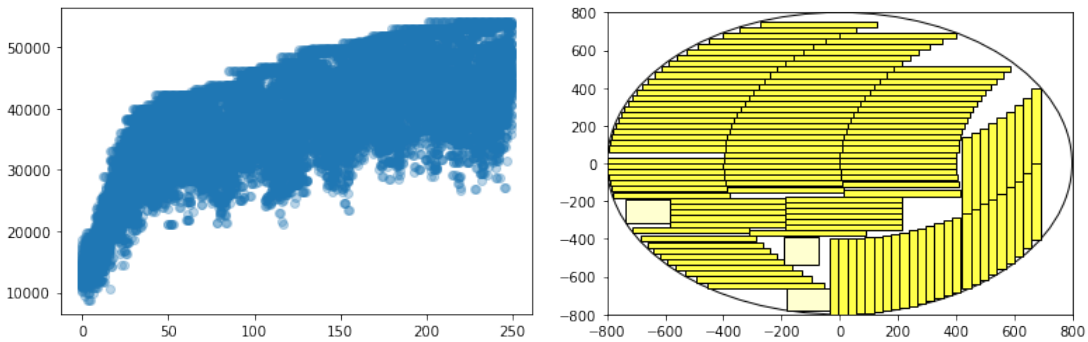
3.4.2. r = 800

Przeprowadziłem dla tego zbioru testy dla 2 zestawów hiperparametrów.

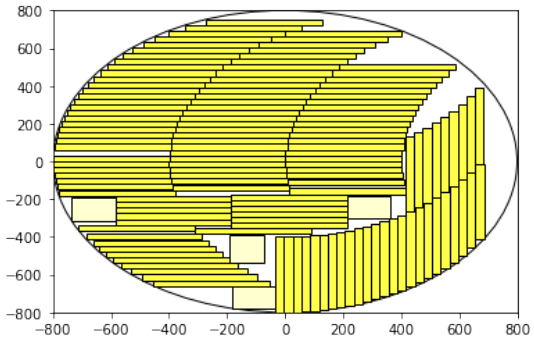
W pierwszym przyjąłem rozmiar populacji równy 100. Liczność elity wybrałem równą 10. Przyjąłem liczbe mutacji 32 oraz krzyżowań 70 w każdej iteracji.



Rys. 3.4. Najlepsze wypełnienie koła w początkowej populacji. Wartość przystosowania wynosi 16740. Tym razem jesteśmy daleko od wymaganej wartości 30000



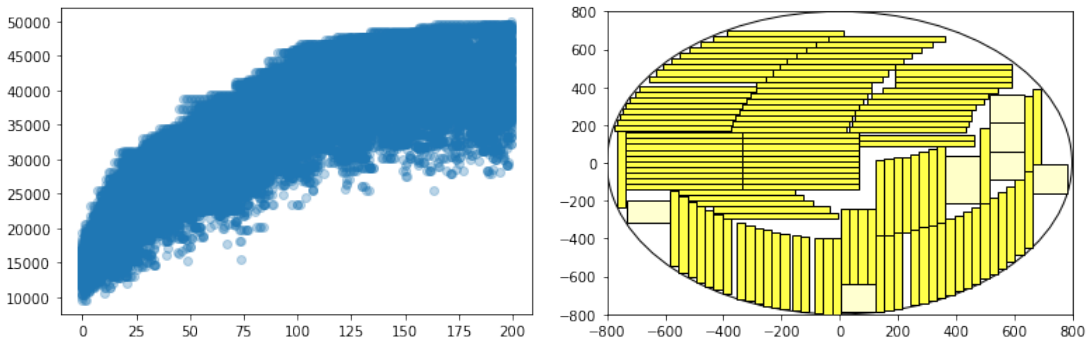
Rys. 3.5. Najlepsze wypełnienie koła po 250 iteracjach oraz wykres wartości przystosowania osobników w czasie (od liczby iteracji). Wartość przystosowania wynosi 54120.



Rys. 3.6. Najlepsze wypełnienie koła po 500 iteracjach. Wartość przystosowania wynosi 54160. Po 250 iteracjach nasz algorytm ma już bardzo małe prawdopodobieństwo na poprawienie wyniku. Przyglądając się osobnikowi możemy jednak zauważyć, że jest możliwe poprawienie go za pomocą nawet pojedynczej mutacji. Programowi nie udawało się jednak wylosować takiej mutacji.

Postanowiłem przetestować także większą populację kosztem zmniejszenia procentowej liczby krzyżowań. W poniższym przykładzie przyjąłem rozmiar populacji równy 200. Liczność elity wybrałem równą 20. Przyjąłem liczbe mutacji 62 oraz krzyżowań 64 w każdej iteracji. Zatem

zwiększyłem populacje dwókrotnie, zachowując procentowy udział elity i mutacji, zmniejszając procentowy udział krzyżowań około 2-krotnie.



Rys. 3.7. Przy 2 razy wiekszej populacji: Najlepsze wypełnienie koła po 200 iteracjach oraz wykres wartości przystosowania osobników w czasie (od liczby iteracji). Wartość przystosowania wynosi 49940.

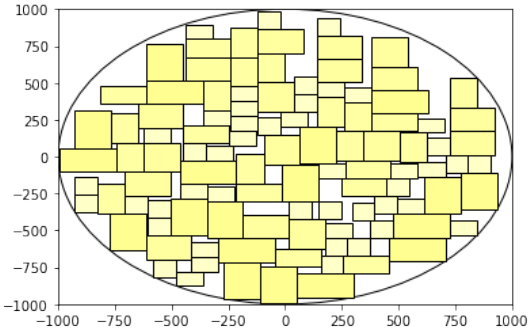
Widzimy, że osiągnięty wzrost przystosowania jest podobny. Zwiększenie populacji spowodowało natomiast wolniejsze działanie algorytmu.

W obu przypadkach wymagany wynik 30000 osiągamy już po około 20 iteracjach i potrafimy osiągnąć wynik dużo lepszy.

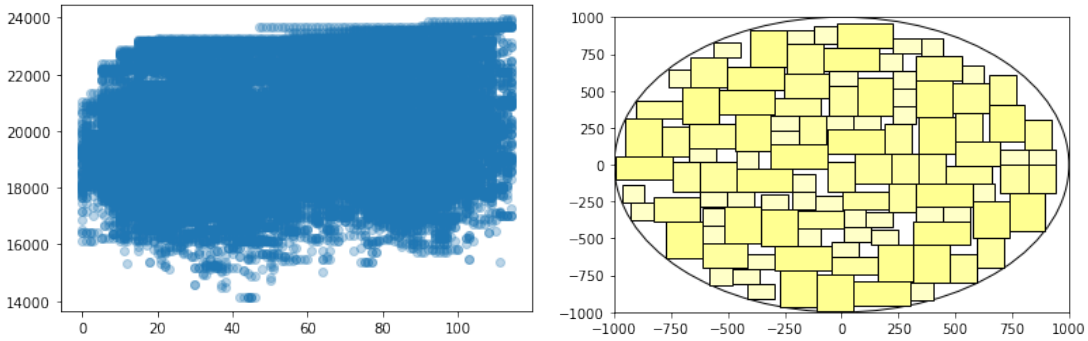
3.4.3. r = 1000

Postanowiłem dla tego zbioru porównać różną liczbę mutacji i krzyżowań. Przetestuję 3 przypadki. W każdym z nich liczność populacji wynosi 100, z czego elita stanowi 10%.

32 krzyżowania, 32 mutacje

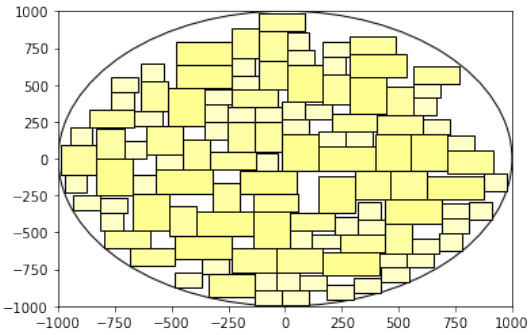


Rys. 3.8. Najlepsze wypełnienie koła w początkowej populacji. Wartość przystosowania wynosi 20880.

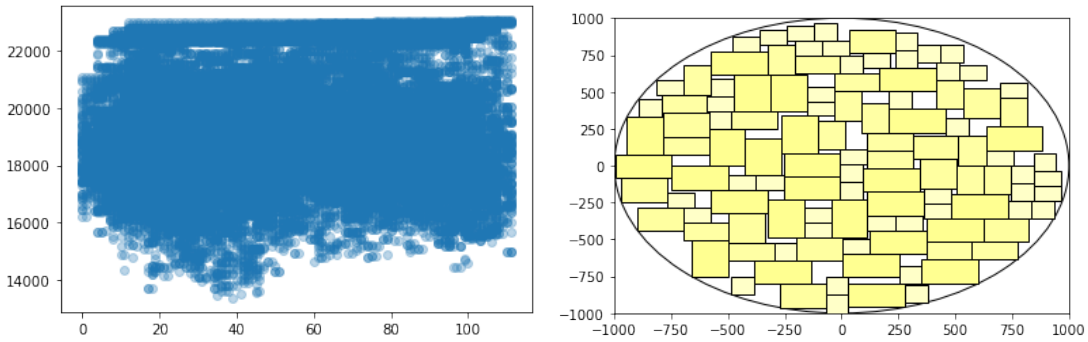


Rys. 3.9. Najlepsze wypełnienie koła po 400 iteracjach oraz wykres wartości przystosowania osobników w czasie (od liczby iteracji). Wartość przystosowania wynosi 23960.

64 krzyżowania, 16 mutacji

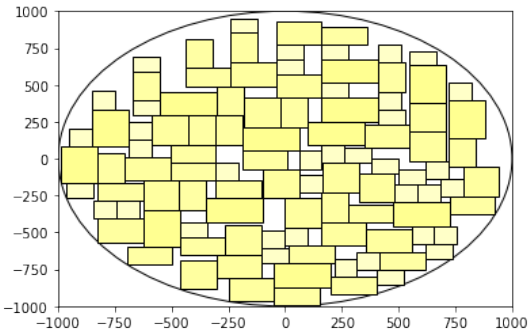


Rys. 3.10. Najlepsze wypełnienie koła w początkowej populacji.

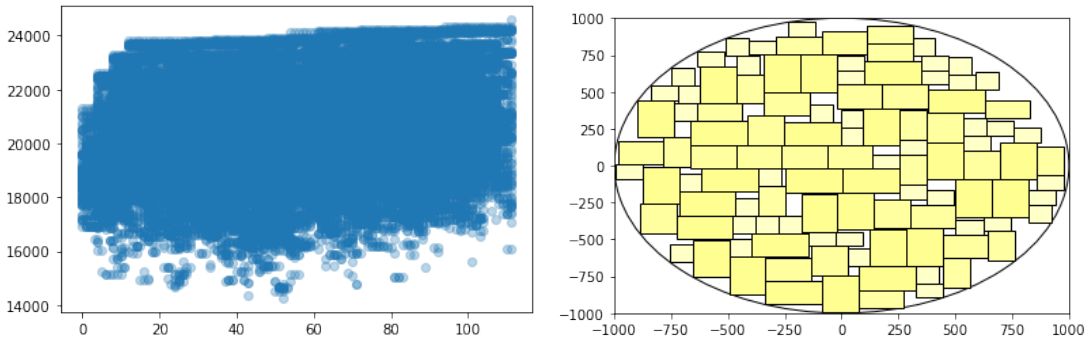


Rys. 3.11. Najlepsze wypełnienie koła po 400 iteracjach oraz wykres wartości przystosowania osobników w czasie (od liczby iteracji). Wartość przystosowania wynosi 23080.

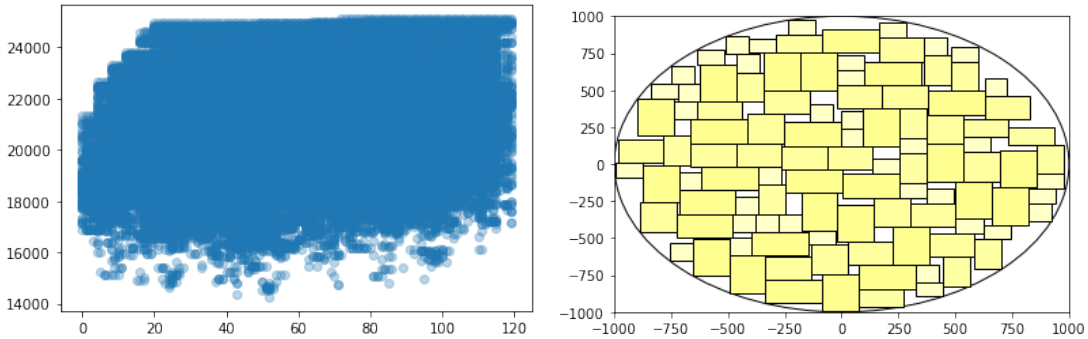
16 krzyżowań, 64 mutacje



Rys. 3.12. Najlepsze wypełnienie koła w początkowej populacji.



Rys. 3.13. Najlepsze wypełnienie koła po 400 iteracjach oraz wykres wartości przystosowania osobników w czasie (od liczby iteracji). Wartość przystosowania wynosi 24580.



Rys. 3.14. Najlepsze wypełnienie koła po 600 iteracjach oraz wykres wartości przystosowania osobników w czasie (od liczby iteracji). Wartość przystosowania wynosi 25100.

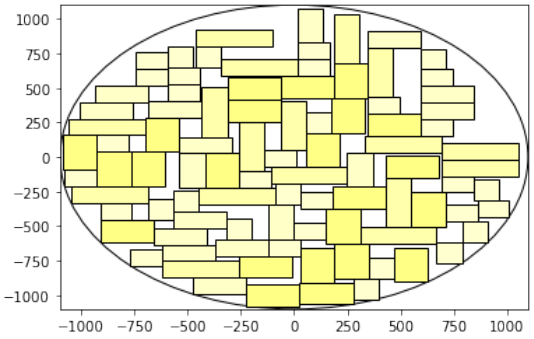
W tym przypadku już trudniej było osiągnąć wymagany próg 25000. Musiałem specjalnie wykonać dodatkowe iteracje na ostatnim przypadku.

Powyższe 3 eksperymenty mogłoby się zdawać, że sugerują, że liczba mutacji ma większe znaczenie od liczby krzyżowań. Jednak są to tylko pojedyncze testy na losowych startowych populacjach, a różnice pomiędzy nimi są niewielkie. Zatem to za mało aby wyciągać takie wnioski. Dużo większe znaczenie na pewno ma start algorytmu - początkowa populacja.

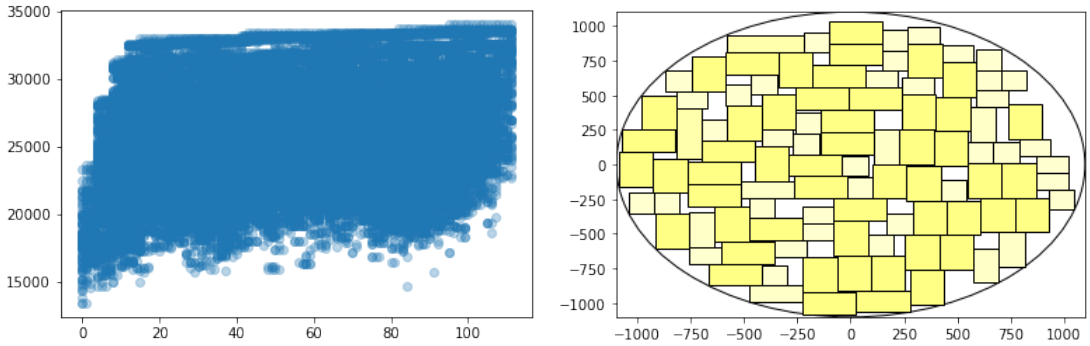
3.4.4. r = 1100

Dla tego zbioru wybrałem następujace parametry:

— rozmiar populacji: 100
— liczba mutacji: 70 (70%)
— liczba krzyżowań: 20 (20%)
— liczność elity: 10 (10%)



Rys. 3.15. Najlepsze wypełnienie koła w początkowej populacji. Wartość przystosowania to 22260.



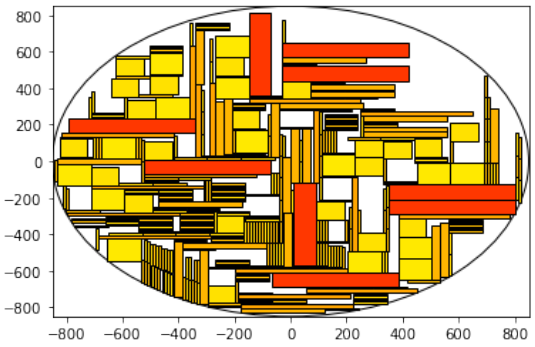
Rys. 3.16. Najlepsze wypełnienie koła po 400 iteracjach oraz wykres wartości przystosowania osobników w czasie (od liczby iteracji). Wartość przystosowania wynosi 34020.

Próg dla tego zbioru wynosił 25000 i także został łatwo osiągnięty.

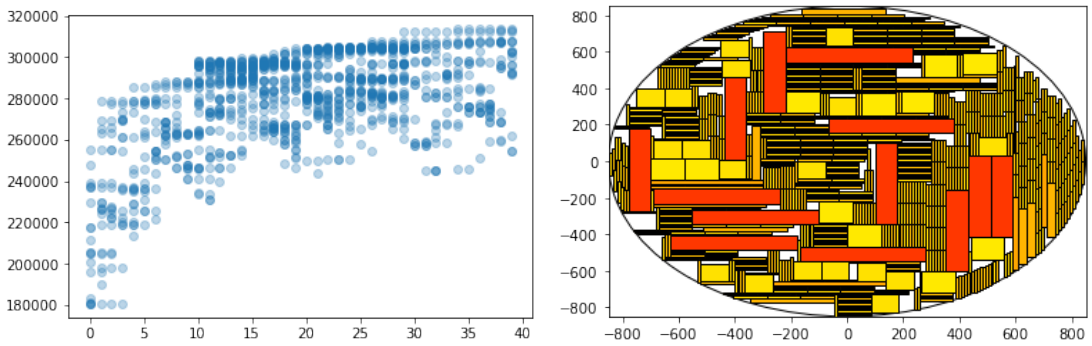
3.4.5. r = 850

W tym podzadaniu nie było określonego progu do przekroczenia. Było w nim najwięcej rodzajów prostokątów (7 rodzajów, gdzie w pozostałych było to zazwyczaj 4, raz 5). Ale przede wszystkim prostokąty są dużo mniejsze i dużo więcej prostokątów mieści się w kole. Algorytm w przypadku tego zadania działał najdłużej. Dlatego zdecydowałem się tutaj na mały rozmiar populacji i mniejszą liczbę iteracji. Wybrałem następujace parametry:

— rozmiar populacji: 20
— liczba mutacji: 14 (70%)
— liczba krzyżowań: 6 (30%)
— liczność elity: 4 (20%)



Rys. 3.17. Najlepsze wypełnienie koła w początkowej populacji. Wartość przystosowania to 230130.



Rys. 3.18. Najlepsze wypełnienie koła po 40 iteracjach oraz wykres wartości przystosowania osobników w czasie (od liczby iteracji). Wartość przystosowania wynosi 313520.

Wykonanie 40 iteracji w przypadku tego zbioru trwało dłużej niż wszystkie pozostałe eksperymenty razem wzięte. Jednak już po nawet tylu operacjach mamy rozwiązanie znacznie lepsze od startowego. Prawdopodobnie daleko temu rozwiązaniu do optymalnego, jest ono mocno uzależnione od rozwiązań w początkowej populacji.

4. Optymalizacja wag w sieci MLP z użyciem algorytmu genetycznego

4.1. Opis zadania

Zadanie polegało na implementacji uczenia sieci MLP z użyciem algorytmu genetycznego. Należało zastosować standardowe operatory krzyżowania i mutacji.

Zaimplementowany algorytm należało przetestować przeprowadzając uczenie na 3 zbiorach: iris, multimodal-large, auto-mpg.

4.2. Rozwiązanie

W rozwiązaniu wykorzystałem przygotowaną wcześniej w ramach laboratorium z sieci neuronowych implementację sieci MLP.

Każdy osobnik przechowuje w sobie sieć neuronową przechowującą warstwy zawierające wagi w postaci macierzy i biasy w postaci wektorów.

Raz powstały osobnik nie jest już modyfikowany, a jedynie powstają jego kopie, zatem ewaluacja może być przeprowadzana podczas jego tworzenia. Ewaluacja jest najbardziej kosztowną obliczeniowo operacją w algorytmie, dlatego wykonywanie jej tylko raz dla każdego osobnika pozwoli przyspieszyć działanie algorytmu.

Ewaluacją jest obliczenie funkcji straty (MSE) na danym zbiorze uczącym. Staramy się je minimalizować.

Jako operator mutacji przyjąłem mutację gaussowską na losowej warstwie sieci, tzn. do każdej wagi w losowo wybranej warstwie dodajemy zmienną z rozkładu normalnego o centrum w zerze i o określonej wariancji.

Jako operator krzyżowania przyjąłem wymianę wag z losowej warstwy, tzn. dziecko powstaje jako kopia pierwszego rodzica, ale w jednej losowo wybranej warstwie wagi dziedziczy od drugiego rodzica.

Sieci osobników inicjalizowane są standardową metodą inicjalizacji sieci wybraną z mojej implementacji sieci, czyli z rozkładu jednostajnego na przedziale $(-1, 1)$ lub metodą *Xavier*. Domyślnie i w eksperymentach stosuję metodę *Xavier*.

Selekcja przeprowadzana jest w dwóch etapach. W pierwszym wybierana jest elita o określonej liczości - osobniki o najlepszej funkcji przystosowania. Następnie przeprowadzana jest selekcja turniejowa. Do osiągnięcia oczekiwanego rozmiaru populacji wyjściowej osobniki wybierane są jako zwycięzcy turniejów. Turniej to `tournament_size` losowo wybranych osobników z dostępnej puli, zwycięzcą jest osobnik o najniższej funkcji ewaluacji.

Rozwiązanie zaimplementowałem w języku Python. Populacja reprezentowana jest przez klasę `Population` udostępniającą następujące istotne dla rozwiązania metody:

— Konstruktor o argumentach:
  — `size` - rozmiar populacji
  — `input_size` - rozmiar wejścia osobników
  — `layers_sizes` - lista rozmiarów warstw (liczby neuronów w nich)

  - `activations` - lista funkcji aktywacji warstw, długość musi być taka sama jak długość `layers_sizes`
  - `x`, `y` - wzorce uczące, odpowiednio macierze zawierające w wierszach wejścia i wyjścia.
  - `initializer` - metoda inicjalizacji sieci osobników. Domyślnie *xavier*.
- iteration - metoda przeprowadzająca pojedynczą iterację ewolucji. Przyjmuje argumenty:
  - `p_mutation` - prawdopodobieństwo mutacji dla każdego osobnika w iteracji
  - `sigma` - wariancja mutacji
  - `p_crossing` - prawdopodobieństwo dla każdego osobnika w iteracji, że weźmie udział w krzyżowaniu jako pierwszy rodzic
  - `elite_count` - liczność elity w selekcji
  - `tournament_size` - rozmiar turnieju w selekcji
- best - osobnik o najlepszej funkcji przystosowania

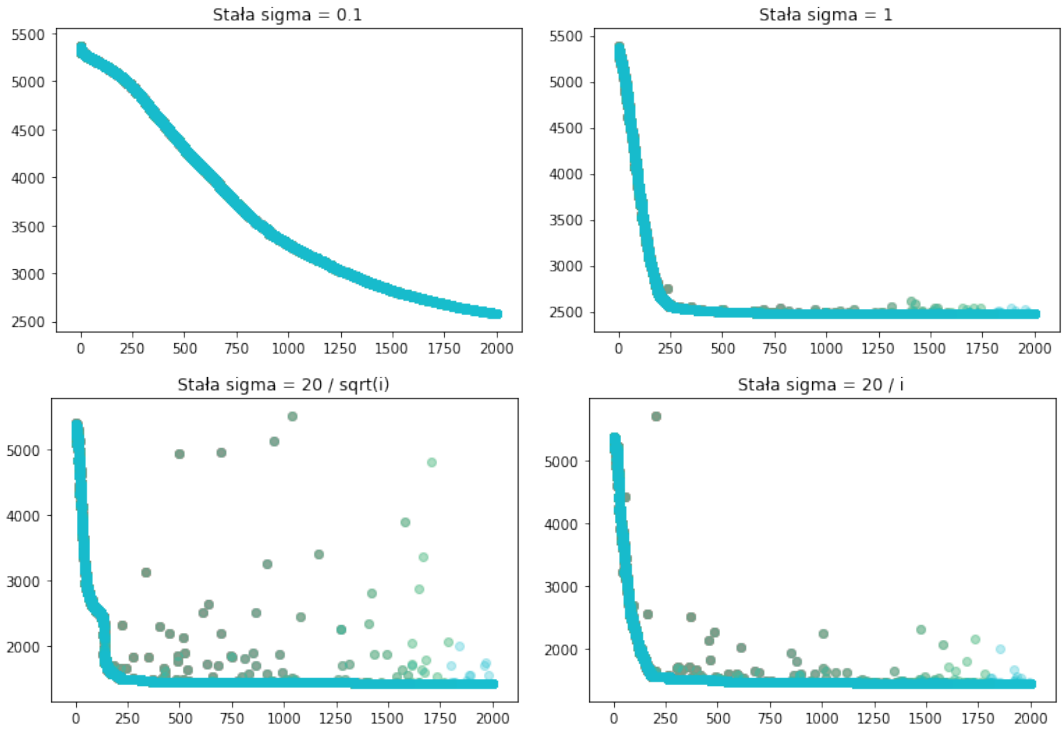
Klasa `Individual` reprezentuje pojedynczych osobników. Możemy policzyć ich MSE za pomocą metody `eval` oraz dostać się do ich sieci przechowywanej w polu `net`.

4.2.1. Testowanie na poszczególnych zbiorach

multimodal-large

Podobnie, jak w pierwszym zadaniu postanowiłem zacząć od przetestowania, jaka wariancja w mutacji daje najlepsze wyniki. W tym celu porównałem 2 stałe wartości wariancji i 2 zmienne w czasie. Każdą próbę przeprowadziłem na 2000 iteracji. Pozostałe argumenty przyjąłem takie same we wszystkich przypadkach:

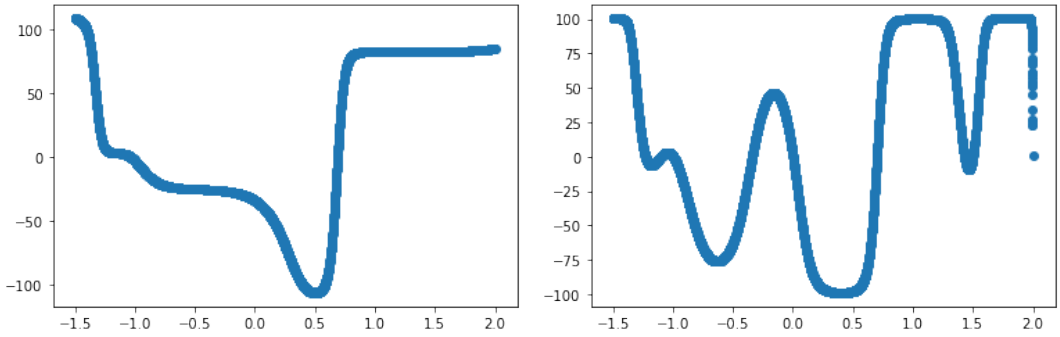
— liczność populacji: 40
— prawdopodobieństwo mutacji: 25%
— prawdopodobieństwo krzyżowania: 50%
— liczność elity: 10 (25% populacji)
— rozmiar turnieju: 4
— architektura sieci: 2 warstwy ukryte po 5 neuronów, z sigmoidalną funkcja aktywacji, na wyjściu funkcja aktywacji liniowa



Rys. 4.1. Wartość MSE osobników populacji w zależności od liczby wykonanych iteracji. Wykresy dla różnych wartości wariancji mutacji.

Widzimy, że sieć na początku uczy się bardzo szybko, a potem utyka. Algorytm nie jest w stanie przekroczyć pewnego poziomu MSE równego około 1440. A wiemy z laboratorium o sieciach neuronowych, że stworzenie sieci o takiej samej architekturze przybliżającej tą funkcję lepiej jest możliwe. Różne wariancje mutacji nie wystarczają aby przekroczyć ten próg. W pierwszym przypadku, kiedy wariancja mutacji była mała algorytm uczył się dużo wolniej.

Zobaczmy jeszcze, jak wygląda predykcja sieci o najlepszym MSE spośród wszystkich, równym X.



Rys. 4.2. Po lewej przybliżenie funkcji multimodal przez sieć uzyskaną algorytmem ewolucyjnym. Po prawej prawdziwy kszałt zbioru

Widzimy, że nie przypomina to właściwego kształtu funkcji. Możemy stwierdzić, że nie udało się nauczyć sieci przy pomocy algorytmu.

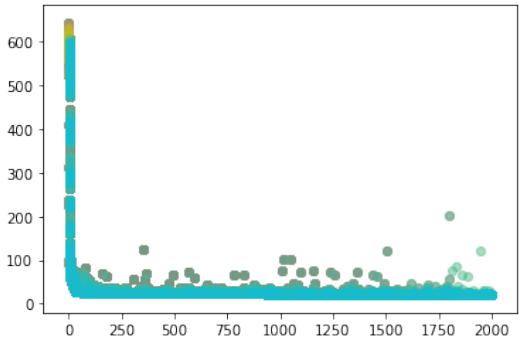
auto-mpg

W tym zbiorze jako zmienną celu traktowaliśmy zmienną `mpg`. W celu uproszczenia wyrzuciłem ze zbioru zmienną kategoryczną.

W przypadku poprzedniego zbioru można było zauważyć małą wariancję przystosowania osobników wewnątrz populacji. Tutaj zwiększyłem liczność populacji, prawdopodobieństwa krzyżowania i mutacji oraz zmniejszyłęm procentowy udział elity. Myślę, że powinno to nieco zwiększyć wariancję przystosowania. Tutaj też zdecydowałem się zastosować wygaszanie wariancji mutacji w czasie. Postanowiłem też przetestować trochę głębszą sieć.

Podsumowując przyjąłem następujące parametry:

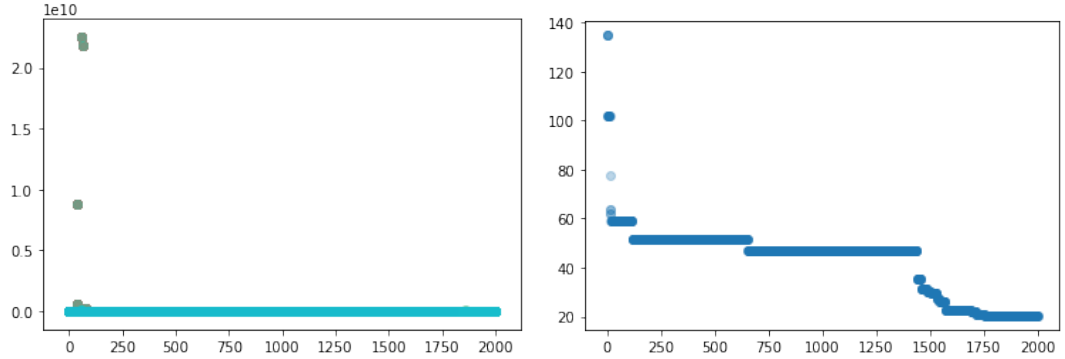
— liczność populacji: 100
— prawdopodobieństwo mutacji: 40%
— wariancja mutacji: $10/log(i)$, gdzie i to numer iteracji
— prawdopodobieństwo krzyżowania: 80%
— liczność elity: 10 (10% populacji)
— rozmiar turnieju: 4
— architektura sieci: warstwy 25+25+25+5+1, ukryte warstwy z sigmoidalną funkcja aktywacji, na wyjściu funkcja aktywacji liniowa



Rys. 4.3. Wartość MSE osobników populacji w zależności od liczby wykonanych iteracji dla zbioru auto-mpg. Sieć z sigmoidalnymi funkcjami aktywacji.

Najlepsze MSE jakie udało się osiągnąć to 21.08.

Porównajmy ten wynik dla wyniku dla takiej samej sieci, ale ze zmienioną funkcją aktywacji we wszystkich warstwach ukrytych na ReLU.



Rys. 4.4. Po lewej wartość MSE wszystkich osobników populacji w zależności od liczby wykonanych iteracji dla zbioru auto-mpg. Po prawej wartość najlepszych osobników. Sieć z ReLU funkcjami aktywacji.

Najlepsze MSE, jakie udało się osiągnąć to 20.4. Jest to wynik bardzo podobny, jak przy poprzedniej architekturze. Nie można powiedzieć, by zmiana funkcji aktywacji poprawiłą coś wynik.

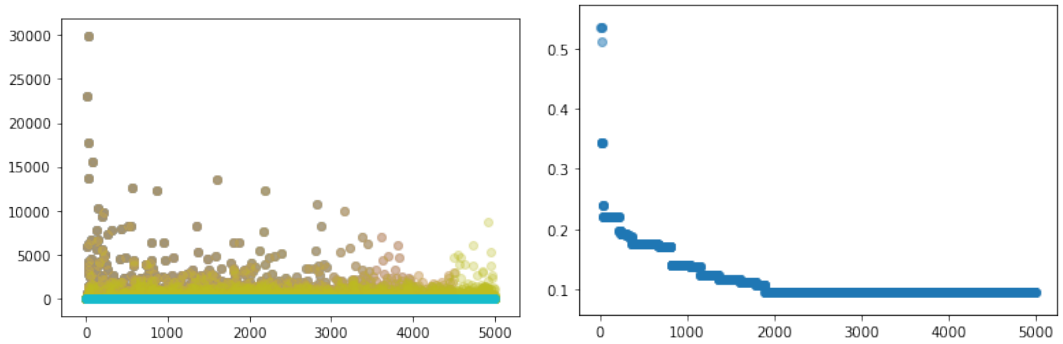
iris

W tym przypadku także potraktowałem ten zbiór jako zbiór do regresji. Przewidywałem pierwszą zmienną (sepal length) na podstawie pozostałych. Tym razem zachowałem zmienne kategoryczne, zakodowane one-hot-encoding.

Przyjąłem podobne parametry, jak przy poprzednim eksperymencie, ale zwiększyłem dwukrotnie rozmiar populacji i zmieniłem trochę architekturę sieci.

— liczność populacji: 200
— prawdopodobieństwo mutacji: 40%
— wariancja mutacji: $10/log(i)$, gdzie i to numer iteracji
— prawdopodobieństwo krzyżowania: 80%
— liczność elity: 20 (10% populacji)
— rozmiar turnieju: 4
— architektura sieci: warstwy 50+50+50+1, ukryte warstwy z funkcją aktywacji ReLU, na wyjściu funkcja aktywacji liniowa

Przeprowadziłem też większą liczbę iteracji ewolucji.



Rys. 4.5. Po lewej wartość MSE wszystkich osobników populacji w zależności od liczby wykonanych iteracji dla zbioru iris. Po prawej wartość najlepszych osobników.

Najlepsze MSE, jakie udało się osiągnąć, to 0.09545. Algorytm nie był w stanie poprawić tej wartości przez ponad 3000 iteracji.