Streszczenie

W poniższym dokumencie przedstawiam moją drogę prób i błędów podczas pisania projektu regresji liniowej z Metod Probabilistycznych w Uczeniu Maszynowym. Ten dokument będzie się składał z dwóch części - w pierwszej wycisnę do maksimum możliwości model nie używając funkcji bazowych - czyli traktując wynik jako kombinację liniową cech. W drugiej części dokumentu spróbuję przeanalizować zależności między cechami i na tej podstawie jak najlepiej dopasować naszą krzywą do punktów

Przygotowania i pierwsze kroki

Projekt zacząłem od napisania funkcji wczytującej dane. Za każdym razem funkcja losuje kolejność danych, które następnie dzieli na zbiory: treningowy, walidacyjny i testowu Stwierdziłem, że na dobry początek odpowiednim podziałem będzie $0.6 : 0.2 : 0.2$, zatem w takiej proporcji początkowo testowałem model. Również początkowo zdecydowałem się użyć kwadratowej funkcji błędu, która dla ustalonych funkcji bazowych liczy połowę średniego kwadratu odległości predykcji od faktycznej wartości.

$$l(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (y^{(i)} - \sum_j \theta_j \phi_j(x^{(i)}))^2$$

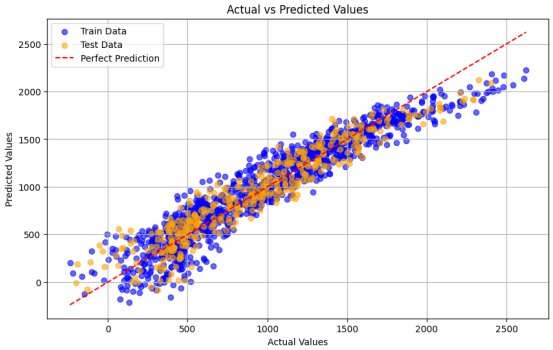
Żeby przeskalować dane zdecydowałem się użyć standaryzacji. Niestety nie jestem wybitnym programistą...

Standaryzacja danych

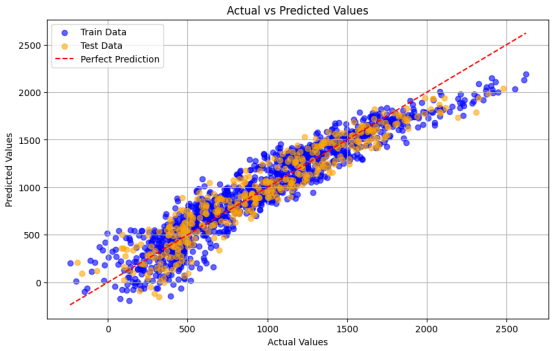
Smutny dla mnie rozdział. Bardzo bolesny. W poniedziałek, 7 kwietnia, po prawie tygodniu prac nad projektem odkryłem przyczynę moich dotychczasowych trudności. Dlaczego nie mogę zbić MSE? Dlaczego `learning rate` jest tak niestabilny numerycznie przy doborze różnych funkcji bazowych? Słuchając znajomych chwalących się swoimi `MSE` zauważyłem ciekawą zależność − ich `MSE` na zbiorze **treningowym** było **dużo niższe**, niż moje rozwiązanie analityczne, mimo użycia tej samej funkcji straty. Przegrzebałem cały ten kod i znalazłem − literówka funkcji odpowiedzialnej za standaryzację danych. Skutkowało to tym, że jak przed standaryzacją model miał trudności z nauczeniem się na danych, tak po niej przestawały one mieć jakikolwiek sens.

Dlaczego standaryzacja danych jest tak ważna ?
Zamiast mierzyć dane wejściowe przez ich wartość, w momencie gdy mamy wystarczająco duży zbiór treningowy lepiej jest mierzyć inną wartość − jaką wielokrotność odchylenia standardowego przyjmuje nasza cecha od jej średniej wartości? Dzięki temu ustalamy wspólną miarę dla każdej z cech.

Niech $T = \{[x_1^{(i)}, x_2^{(i)}, \ldots, x_7^{(i)}, y^{(i)}]\} = \{(x^{(i)}, y^{(i)})\}$ będzie naszym zbiorem treningowym, Definiuję następujące przekształcenie standaryzujące $s : T \to S$ w taki sposób, że $s(x^{(i)})_j = \frac{x_k^{(i)} - \mathbb{E}[x_j]}{\sqrt{\mathbf{Var}(x_j)}}$ oraz $s(y^{(i)}) = y^{(i)}$, gdzie wartość oczekiwana i wariancja jest brana względem rozkładu jednostajnego nad T . W ten sposób tworzę zbiór ustandaryzowanych danych S, na którym będę trenować mój model. W momencie gdy będę mierzył skuteczność mojego modelu na zbiorze testowym (lub wykorzystywał zbiór walidacyjny), to dane znajdujące się w nim będę mierzył miarą mojego **zbioru treningowego** - będę przekształcał używając policzonych na nim średnich i odchyleń standardowych.



Rysunek 1: Dane oryginalne, MSE: 13607.09



Rysunek 2: Dane ustandaryzowane, MSE: 13675.66

Standaryzacja jednak w klasycznym modelu regresji liniowej nie ma aż tak dużego znaczenia - ta której użyłem jest przekształceniem liniowym danych wejściowych, zatem nasz model może się nauczyć innych parametrów co doprowadzi go do podobnych rezultatów. Powyżej przedstawiam porównwanie wyników rozwiązania analitycznego na zbiorze testowym danych oryginalnych i ustandaryzowanych.

Standaryzacja danych użyteczna jest jednak w momencie, gdy będziemy chcieli szerzej je analizować. Gdy ustandaryzujemy nasze dane, możemy na podstawie rozwiązania problemu regresji liniowej wnioskować (po rozmiarze parametrów) od jakich cech nasz wynik zależy najbardziej. Dodatkowo na ustandaryzowane dane możemy swobodnie aplikować funkcje bazowe i regularyzacje, jednocześnie nie przejmując się tym że skala konkretnych cech przyćmi pozostałe.

Algorytmy

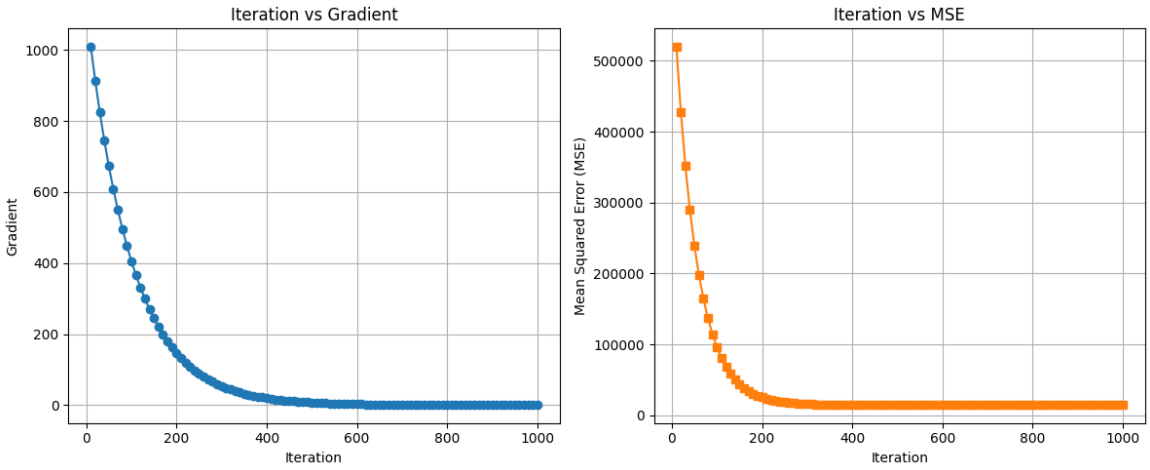
Podczas tego projektu zaimplementowałem dwie wersje algorytmu spadku wzdłuż gradientu. Pierwsza używała pełnego zestawu danych treningowych - jedna iteracja to jedna epoka. W drugiej użyłem metody gradientu mini-batch. Przy **poprawnie** ustandaryzowanych danych odpowiednim `learning rate` okazał się być $\eta = 0.01$ − po prostu gradient przy doborach różnych funkcji bazowych zbiegał odpowiednio szybko, ale nie był na tyle duży żeby algorytm przeskakiwał minimum czy wystrzeliwał do nieskończoności. Algorytm (de facto jeden, w dwóch trybach) miał różne warunki stopu − można ustalić czy uzna model wytrenowany po k iteracjach, czy w momencie gdy norma gradientu spadnie poniżej ustalonej liczby μ.

Na zajęciach wyprowadziliśmy fakt, że jeśli θ minimalizuję funkcję błędu z normalizacją grzbietową o parametrze λ, to $\theta = (X^T X + \lambda I)^{-1} X^T y$, przy czym odwrotność tej macierzy zawsze istnieje dla $\lambda > 0$. Dlatego napisałem również algorytm wyznaczający rozwiązanie analityczne, przy czym macierz $X^T X$ nie zawsze jest odwracalna więc żeby wyznaczyć rozwiązanie bez normalizacji biorę $\lambda \to 0$. Poniżej zamieszczam wyniki otrzymane na zbiorze testowym za pomocą rozwiązania analitycznego dla różnych λ (biorąc jako zbiór treningowy 60% danych wejściowych)

W dalszej części tej sekcji chciałbym porównać skuteczność algorytmu **mini-batch** w porównaniu do klasycznego algorytmu regresji liniowej. Na początku sprawdźmy, jak sprawuje się klasyczny algorytm. Poniższe wykresy przedstawiają zmianę gradientu oraz MSE w kolejnych iteracjach. W tym wywołaniu dostałem MSE na zbiorze treningowym: 14124.78, na testowym: 14630.03, a gradient zbiegł do 0.04.

λ	Train Set MSE	Test Set MSE
1×10^{-6}	14358.6917	14980.0649
0.1	14358.6957	14980.8088
0.2	14358.7079	14981.5606
0.5	14358.7931	14983.8635
1	14359.0972	14987.8599
2	14360.3111	14996.4448
5	14368.7628	15026.8995
10	14398.6450	15093.0902
20	14515.9055	15281.6319
50	15294.8474	16264.2287
100	17821.7851	19099.6548
1000	135659.7145	139351.7539
100000	590708.4282	595069.9019

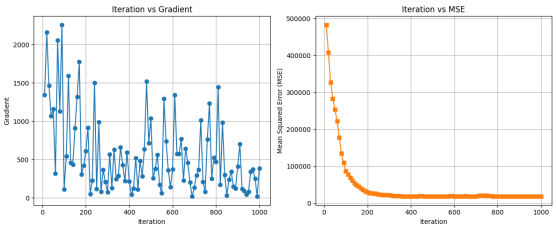
Tabela 1: MSE rozwiązania analitycznego dla różnych wartości λ



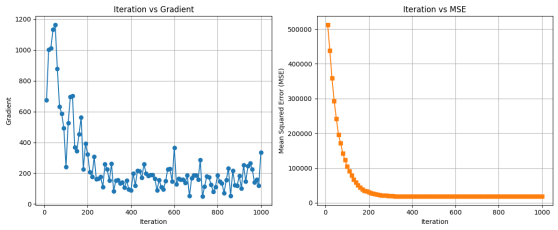
Rysunek 3: Gradient i MSE dla kolejnych iteracji, bez **mini-batch**

Jeśli chodzi o algorytm **mini-batch**, to ma on pewne ciekawe cechy które odkryłem podczas jego używania. Przez to że wybieramy dane w losowy sposób, to próbka którą dostajemy `względnie dobrze` symuluje średnią po wszystkich danych testowych. Jednak jest on ekstremalnie szybszy (miałem sytuacje gdy mini-batch trenował model podobnie efektywnie co zwykły gradient descent, ale robił to w 10 sekund zamiast w 2 minuty). Poniżej porównanie kilku przebiegów algorytmu **mini-batch** dla różnych ustawień rozmiaru pakietu.

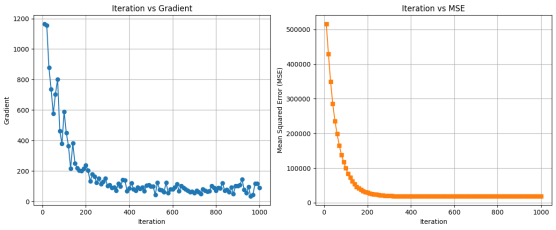
Przez to, że w każdym przebiegu iteracji losujemy podzbiór danych na których się uczymy, to gradient w odróżnieniu do wersji bez **mini-batch** nie zbiega do 0, tylko drga zależnie od wylosowanego podzbioru. Nie przeszkadza to jednak w tym, aby nasz model poprawnie dopasowywał dane. Żeby zrównoważyć efektywność **mini-batch**, wybrałem `batch-size`$= 32$ do dalszych testów.



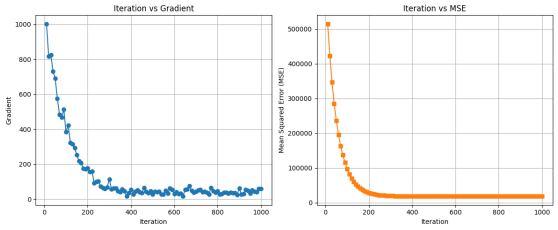
Rysunek 4: `batch size`= 1, `MSE`= 14957.47



Rysunek 5: `batch size`= 8, `MSE`= 14740.81



Rysunek 6: `batch size`= 32, `MSE`= 14584.44

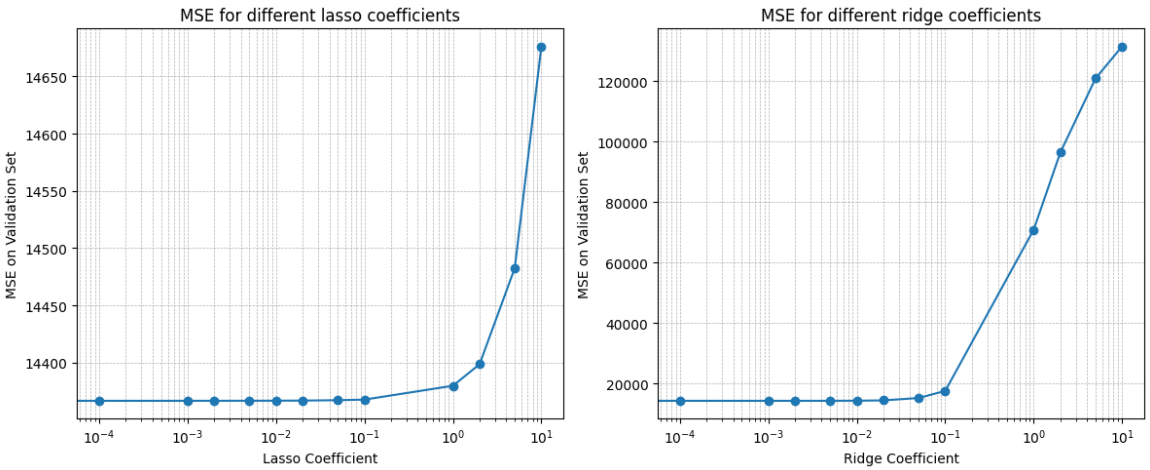


Rysunek 7: `batch size`= 128, `MSE`= 14609.96

Regularyzacja i wyznaczanie hiperparametrów

Podczas trenowania modelu uwzględniłem również regularyzacje L1 i L2 oraz regularyzację z siecią elastyczną. Ąby wyznaczyć parametry regularyzacji, używam metody gradientu w wersji bez **mini-batch**, gdyż różnice dla różnych parametrów są często marginalne. Po rozdzieleniu danych na zbiór treningowy, walidacyjny i testowy porównuję wyniki na zbiorze walidacyjnym dla różnych parametrów $\lambda_1, \lambda_2 \in \{0, 0.0001, 0.001, 0.002, 0.005, 0.01, 0.02, 0.05, 0.1, 1, 2, 5, 10\}$.

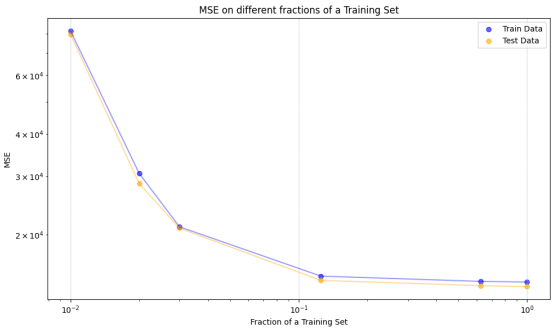
Okazuje się że gdy nie rozważamy funkcji bazowych to regularyzacja nie daje prawie żadnej poprawy gdy chodzi o przewidywanie naszych danych, gdyż nawet na danych treningowych ma problemy z dopasowaniem odpowiedniej płaszczyzny i nie zachodzi przeuczenie. Gdy będziemy jednak trenować nasz model dla wybranych funkcji bazowych, będziemy z niej korzystać żeby algorytm mógł dopasowywać się do danych bez overfittingu. Na wykresach poniżej widzimy, że model póki co osiąga najlepsze wyniki, gdy w ogóle nie korzystamy z regularyzacji.



Rysunek 8: Wyniki dla różnych parametrów regularyzacji na zbiorze walidacyjnym

Uczenie na frakcjach zbioru treningowego

W tym rozdziale chciałbym przeanalizować, jak model poradzi sobie na różnych frakcjach zbioru treningowego. Analogicznie jak we wcześniejszej części dokumentu, będę stosował podział danych na zbiory w proporcji 0.6 : 0.2 : 0.2. Dane które zebrałem zostały przygotowane w następujacy sposób: Mój algorytm miał 5 przebiegów w których za każdym razem najpierw podzieliłem dane na zbiory, a potem odpaliłem algorytm regresji liniowej na różnych frakcjach zbioru treningowego i zebrałem wyniki. Na koniec uśredniłem wyniki dla każdej frakcji.



MSE Results by Training Fraction

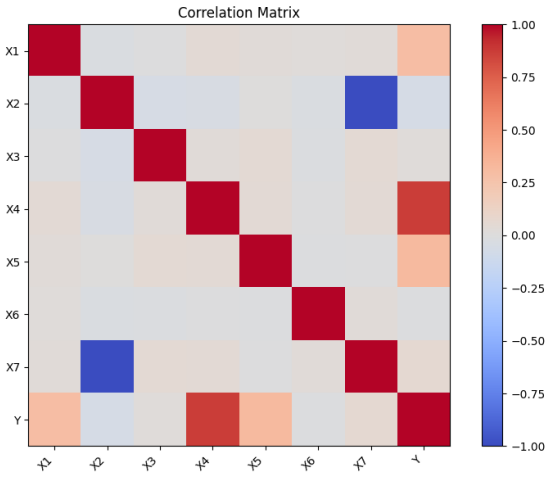
Fraction	Train MSE	Test MSE
0.010	81413.4681	79485.9070
0.020	30508.0960	28445.2618
0.030	21142.7902	20949.3385
0.125	15055.4355	14620.9795
0.625	14520.2944	14097.0583
1.000	14455.3558	14003.4233

Rysunek 9: Wyniki dla różnych frakcji zbioru treningowego

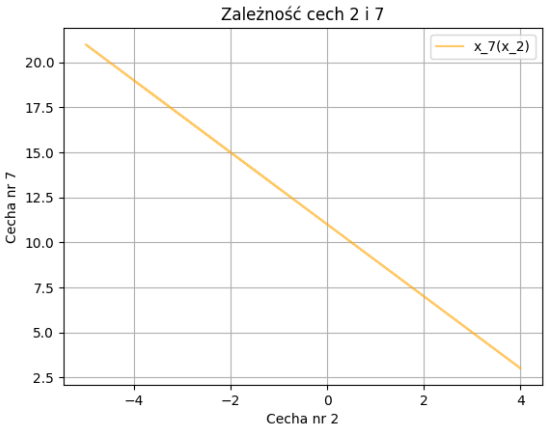
Widzimy, że mimo początkowo ogromnej różnicy model radzi sobie nienajgorzej już, gdy korzystamy z 12.5% czy 62.5% zbioru treningowego. Spróbujmy teraz zbić nasz wynik za pomocą wykorzystania funkcji bazowych.

Szersza analiza danych

Pierwszą rzeczą, od której zacząłem, było stworzenie macierzy korelacji (Pearsone'a) między cechami x oraz wynikiem y na podstawie danych. Macierz ta mówi nam o stopniu zależności między poszczególnymi cechami i wynikiem. Dodatnia korelacja będzie nam mówić o tym że gdy jedna wartość rośnie to druga też, a ujemna, że gdy jedna wartość rośnie, to druga maleje. Poniżej przedstawiam tą macierz w formie heatmapy.

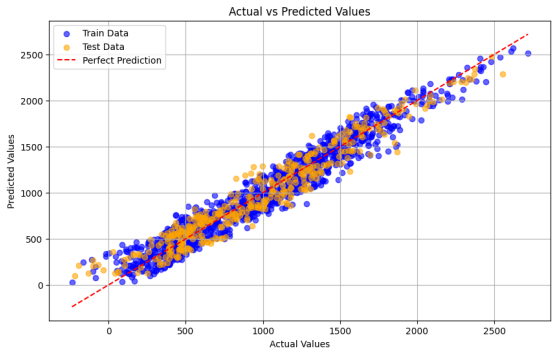


Rysunek 10: Macierz korelacji

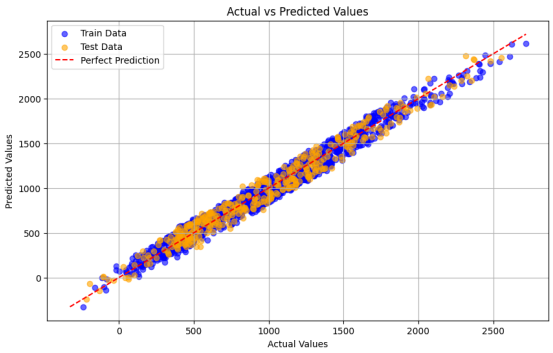


Rysunek 11: Zależność x_2 i x_7

Pierwsze, co bardzo się rzuca w oczy to to jak silnie skorelowane są cechy nr 2 i 7. Okazuje się, że w dostarczonych danych x_7 jest funkcją x_2, zatem jedna z tych cech jest nadmiarowa. Widzimy również, że wynik y jest najbardziej skorelowany z cechami x_1, x_4, x_5, trochę słabiej z x_7 i x_3. Zatem moim strzałem będzie użycie funkcji bazowych $[x_1, x_3, x_4, x_5, x_6, x_7, x_1^2, x_4^2, x_5^2, x_1 \cdot x_4, x_1 \cdot x_5, x_4 \cdot x_5]$. Gradient zbiegał o wiele wolniej, więc puściłem algorytm na 5000 iteracji. Otrzymałem `Train Set MSE = 8069.05` oraz `Test Set MSE = 8945.86`. Jednak używając funkcji bazowych $[W(x) : \deg(W) \leqslant 2]$ otrzymuję jeszcze lepsze MSE, `Train Set MSE = 4356.73, Test Set MSE = 4403.39`. Używając funkji bazowych $[W(x) : \deg(W) \leqslant 3]$ dostaję nieznaczną poprawę, `Train Set MSE: 3836.69, Test Set MSE: 4142.41`



Rysunek 12: Wyniki pierwszej partii funkcji bazowych



Rysunek 13: Wyniki trzeciej partii funkcji bazowych

Cichym bohaterem okazał się tu być algorytm **mini-batch**, który znacząco poprawia szybkość działania algorytmu bez pogarszania jakości uczenia się. Widzimy zatem, że raczej nie chodzi o wielomiany trzeciego stopnia, więc spróbujmy dołożyć coś do tych drugiego.

```
959.72 1
458.77 x[4]
159.75 x[5]
133.77 x[1]
106.43 x[4]x[5]
86.93 x[1]x[3]
34.37 x[4]x[4]
-16.01 x[1]x[1]
5.99 x[3]x[3]
5.71 x[7]
-5.71 x[2]
-3.04 x[3]
-2.69 x[1]x[6]
-2.37 x[1]x[5]
2.24 x[5]x[7]
-2.24 x[2]x[5]
-2.11 x[6]x[6]
1.79 x[4]x[6]
```

Rysunek 14: Współczynniki θ przy kolejnych funkcjach bazowych

Wróćmy do przypadku z wielomianami 2 stopnia i przeanalizujmy parametry przy kolejnych funkcjach bazowych. Widzimy, że model szacuje że największą rolę grają kombinacje

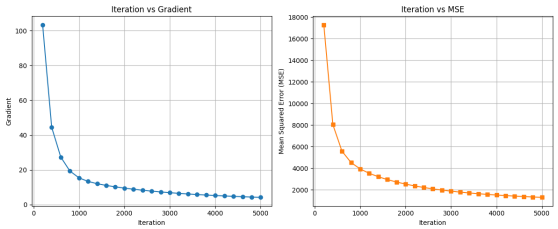
x_1, x_2, x_4, x_5 ale również x_3^2, którego nie uwzględniliśmy przy naszym pierwszym pomyśle. Wysokie współczynniki przy funkcjach liniowych dają mi pomysł, że warto spróbować do naszych cech dołożyć funkcje gaussowskie.

Przetestowałem swoją hipotezę używając różnych funkcji bazowych postaci

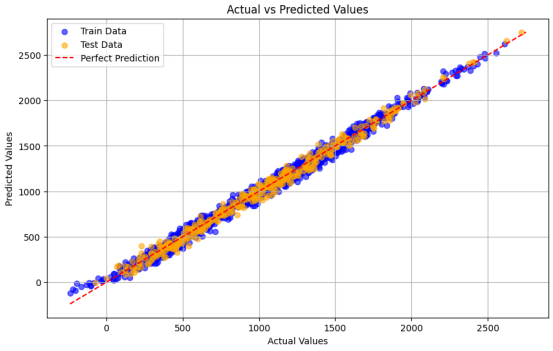
$$[W(x) : \deg(W) \leqslant 2] \cup \bigcup_{i,\alpha}[g(x_i, \alpha)]$$

Gdzie $g(x, \alpha) = \mathrm{e}^{-\frac{x^2}{\alpha^2}}$, dla każdej z nich w oparciu o zbiór walidacyjny wyznaczyłem parametry regularyzacji z siecią elastyczną. Trenowanie tego modelu wraz z wyznaczaniem hiperparametrów zaczęło schodzić wykładniczo długo (uznałem że użycie regularyzacji będzie przydatne, gdyż model zaczął overfitować się do zbioru treningowego). Jednak udało mi się w ten sposób znacząco zbić błąd i w końcu dotrenowałem model do wyników `Train Set MSE: 1291.55 Test Set MSE: 1294.79`. A oto funkcje bazowe których użyłem

$$[W(x) : \deg(W) \leqslant 2] \cup \bigcup_{i\neq 2}[g(x_i, 0.01)] \cup \bigcup_{i\neq 2}[g(x_i, 0.5)]$$



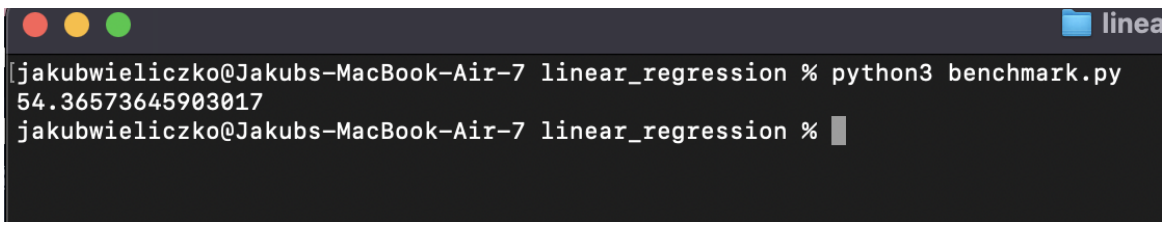
Rysunek 15: Krzywa uczenia ostatecznego modelu



Rysunek 16: Wynik ostateczny

Podsumowanie

Podczas pracy z załączonymi danymi udało mi się zbliżyć do błędu ≈ 1294.79. Po turbulencjach związanych z umiejętnościami kodowania udało mi się również przetestować moje rozwiązanie na konkursie



Rysunek 17: Benchmark jaki uzyskałem