# Miniprojekt I

## Metody probabilistyczne w uczeniu maszynowym

Łukasz Trzos

# 0 Temat projektu

Każdy wiersz pliku `dane.data` zawiera osiem liczb rzeczywistych: pierwsze siedem odpowiada wartościom pewnych cech (danym wejściowym), natomiast ostatnia liczba odpowiada danej objaśnianej (danej wyjściowej).

Zaimplementuj (w ulubionym języku programowania) algorytm **regresji liniowej** i wyznacz funkcję jak najlepiej opisującą dane wyjściowe w zależności od siedmiu cech. Skorzystaj zarówno z rozwiązania analitycznego, jak i metod gradientowych.

Do ewaluacji otrzymanych modeli wykorzystaj **kwadratową funkcję straty**. Dodatkowo możesz sprawdzić, jak zmieniają się wyniki przy wyborze innych funkcji strat (np. rozważając bezwzględną funkcję straty lub funkcję Hubera).

Sprawdź, jak zastosowanie **funkcji bazowych** wpływa na wynik. Przetestuj funkcje bazowe takie jak wielomiany (np. $x_1, x_1^2, x_1 \cdot x_4, \dots$), funkcje gaussowskie (czyli funkcje postaci $x \mapsto \exp\left(-\frac{(x-\bar{x})^2}{s^2}\right)$ z pewnym parametrem $s$) i ewentualnie inne ciekawe funkcje, które mogą poprawić wynik.

Zastosuj **regularyzację** $\ell_1$ i $\ell_2$ oraz regularyzację z siecią elastyczną i w każdym przypadku wyznacz możliwie najlepszą wartość parametru/parametrów regularyzacji.

Nie zapomnij o tym, aby na początku **przeskalować dane**.

Podziel (w sposób losowy) dane na **zbiór treningowy, walidacyjny i testowy**. Hiperparametry modelu (czyli np. parametry regularyzacji, stopień wielomianów, parametr funkcji gaussowskiej) wyznacz w oparciu o dane ze zbioru walidacyjnego, natomiast ocenę modelu oprzyj na danych ze zbioru testowego.

Dla wybranych modeli stwórz **wykresy** zawierające krzywe uczenia, czyli przedstawiające funkcję błędu zależną od rozmiaru zbioru treningowego. Zaznacz punkty odpowiadające błędom obliczonym na całym zbiorze testowym po zastosowaniu algorytmu na następujących frakcjach zbioru treningowego: $0.01, 0.02, 0.03, 0.125, 0.625, 1$. Aby uwiarygodnić wyniki, uśrednij kilka przebiegów algorytmu na losowych wyborach obserwacji do zbioru treningowego, walidacyjnego i testowego.

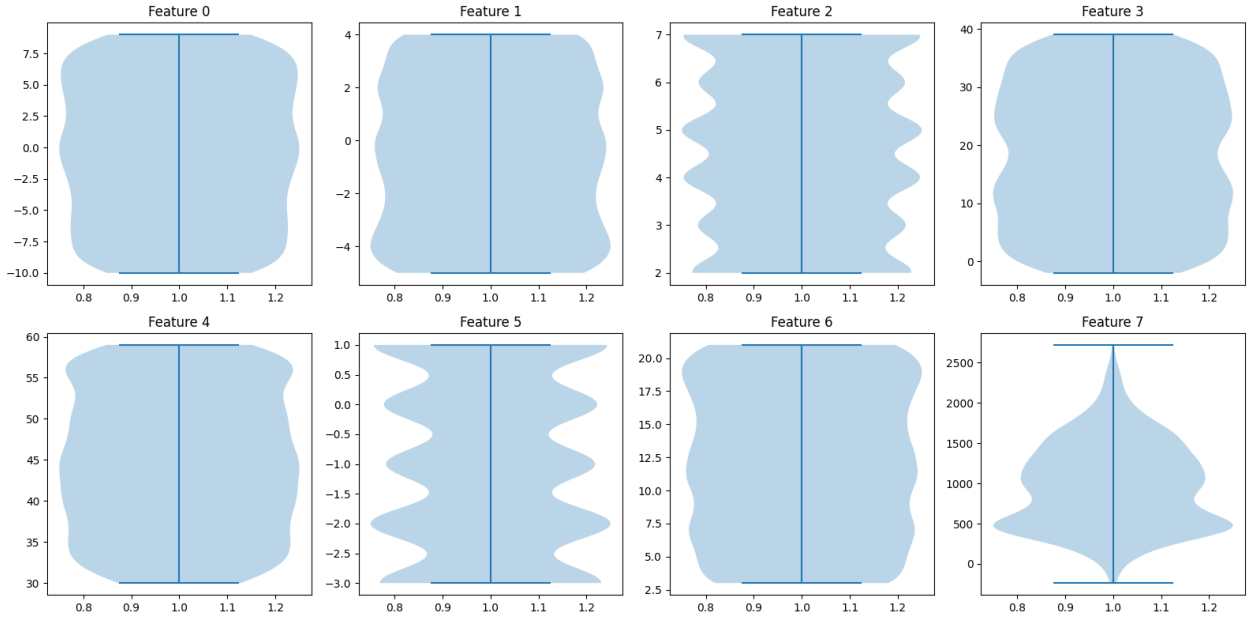
Napisz **raport** opisujący rozważane modele oraz otrzymane wyniki. W opisie uwzględnij wszystkie interesujące aspekty, takie jak podział danych, sposób ich skalowania, wykorzystaną funkcję błędu, wstępną analizę danych, wykorzystane funkcje bazowe, metody regularyzacji, szybkość uczenia itd. Nie musisz się ograniczać tylko do najlepszych wyników – opisywanie porażek również jest cenne.

# 1 Preprocessing danych

Na początku dzielimy dane ze względu na reprezentowane przez nie cechy. Chcielibyśmy każdą cechę przeskalować do przedziału $[0, 1]$. Musimy sprawdzić, czy dane są w rozsądnym zakresie, oraz czy nie występują wśród nich wartości ekstremalne, będące przeciwwskazaniem do użycia takiej metody skalowania

Dla każdej cechy utworzymy wykres ilustrujący jej wartości minimalne oraz maksymalne w naszym zbiorze, oraz rozkład danych między tymi dwoma wartościami. Wykresy utworzymy za pomocą biblioteki matplotlib.

Wykres "Feature 7" odpowiada rozkładowi danej objaśnianej, pozostały wykresy ilustrują kolejne cechy




Rysunek 1: Wykresy ilustrujące rozkłady poszczególnych cech

Jak widać, dane zachowują się dobrze, nie ma punktów znacząco odbiegających wartością od średniej. Zastosujemy więc skalowanie min-max. Dla każdej wartości cechy v, jej nową wartością będzie:

$$v_{scaled} = \frac{v - v_{min}}{v_{max} - v_{min}}$$

gdzie $v_{max}$ i $v_{min}$ oznaczają odpowiednio maksymalną i minimalną wartość danej cechy, która wystąpiła w otrzymanym zbiorze danych. W ten sposób wszystkie nowe dane będą znajdować się w przedziale $[0, 1]$.

# 2 Podstawowy model

## 2.1 Metoda analityczna

Na początek pracy z przygotowanymi danymi stworzymy prosty model, w którym:

- Wykorzystamy kwadratową funkcję straty
- Nie uwzględnimy parametru regularyzacji
- Rozwiązanie wyznaczymy analitycznie

Dane będziemy dzielić na zbiór treningowy, walidacyjny i testowy w proporcji $0.7, 0.15, 0.15$. Ponieważ w tym konkretnym modelu nie mamy żadnych hiperparametrów, zbiór walidacyjny możemy wykorzystać jako rozszerzenie zbioru treningowego. Rozwiązanie analityczne dla problemu regresji z kwadratową funkcją straty i bez regularyzacji jest postaci:

$$\theta = (X^T X)^{-1} X^T y$$

W każdej iteracji wykorzystamy więc odpowiednią część zbioru treningowego do obliczenia optymalnego wektora $\theta$, następnie na podstawie danych ze zbioru testowego obliczymy uzyskany błąd:

$$e = \frac{1}{m} \sum_{i=1}^{m} (y_{test_i} - X_{test_i}\theta)^2$$

Do obliczeń wykorzystamy następujący kod:

```python
def SolveForTheta():

    for _ in range(NUMBER_OF_TOTAL_TRIES):

        #Get the datasets
        trainingSet, validatingSet, testingSet = ds.GetDataSplit(0.85,0,0.15)

        X_test, y_test = ds.CreateMatrices(testingSet)

        for part in trainingSetSetParts.keys():

            #Create a subset of the training dataset
            elementNumber = int(len(trainingSet) * part)

            trainingSetSubset = trainingSet[:elementNumber]

            #Analytical method of solving
            X, y = ds.CreateMatrices(trainingSetSubset)
            theta = np.linalg.inv(X.T @ X) @ (X.T @ y)

            #Calculate the error
            y_pred = X_test @ theta

            quadraticError = np.sum((y_pred - y_test) ** 2)/len(testingSet)
            trainingSetSetParts[part].append(float(quadraticError))

    #Print results
    for (key, value) in trainingSetSetParts.items():
        print(f"Average error for {key} part of the trainingSet set was {np.mean(value)}")
```

Rysunek 2: Kod podstawowego modelu

Wyniki przedstawia wykres:


Rysunek 3: Krzywa uczenia się dla modelu podstawowego

Średnia wartość wektora $\theta$ wśród próbek, w których wykorzystaliśmy cały zbiór treningowy wyniosła:

$$\theta^T = [0.151, -0.018, -0.009, 0.529, 0.177, 0.002, -0.010]$$

## 2.2 Metoda gradientowa

Wprowadzimy lekkie zmiany do podstawowego modelu. Parametr $\theta$ nie będzie już wyznaczany za pomocą wzoru, będziemy go obliczać algorytmem iteracyjnym. Chcemy zminimalizować kwadratową funkcję straty na zbiorze treningowym, której wartość wynosi $J(\theta) = \frac{1}{m}(X\theta - y)^T(X\theta - y) = \frac{1}{m}(\theta^T X^T X\theta - 2\theta^T X^T y + y^T y)$. Obliczamy pochodną:

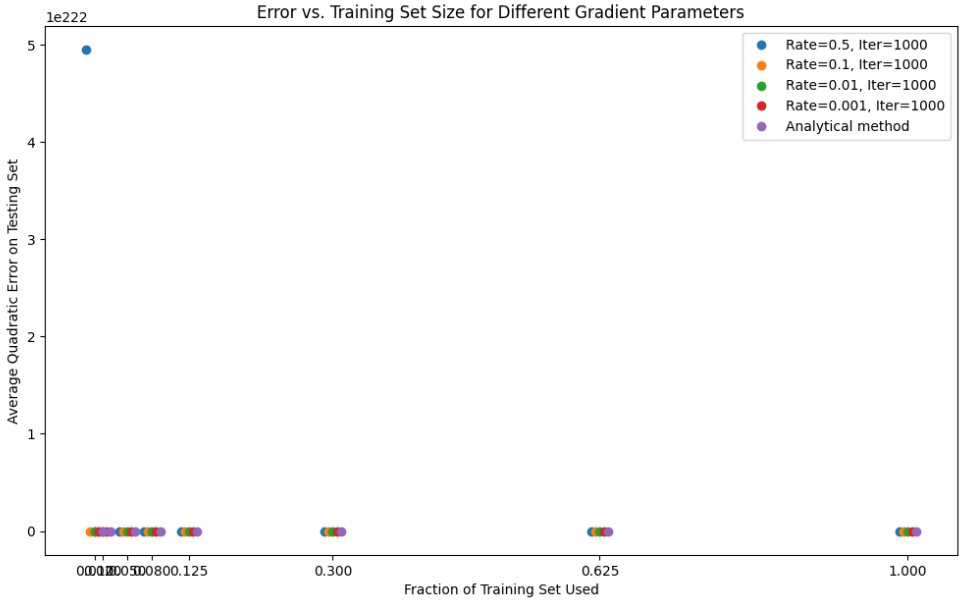
$$\frac{dJ(\theta)}{d\theta} = \frac{1}{m}(2X^T X\theta - 2X^T y) = \frac{2}{m}X^T(X\theta - y)$$

Chcąc zminimalizować wartość funkcji straty będziemy przesuwać $\theta$ w przeciwną stronę niż wskazuje obliczony gradient. Jako wektor startowy przyjmiemy $\theta = 0$. Ważnymi parametrami są współczynnik spadku względem gradientu, oraz warunek stopu. Możemy zbadać jak zmienia się obliczony błąd na podstawie wyboru tych parametrów.

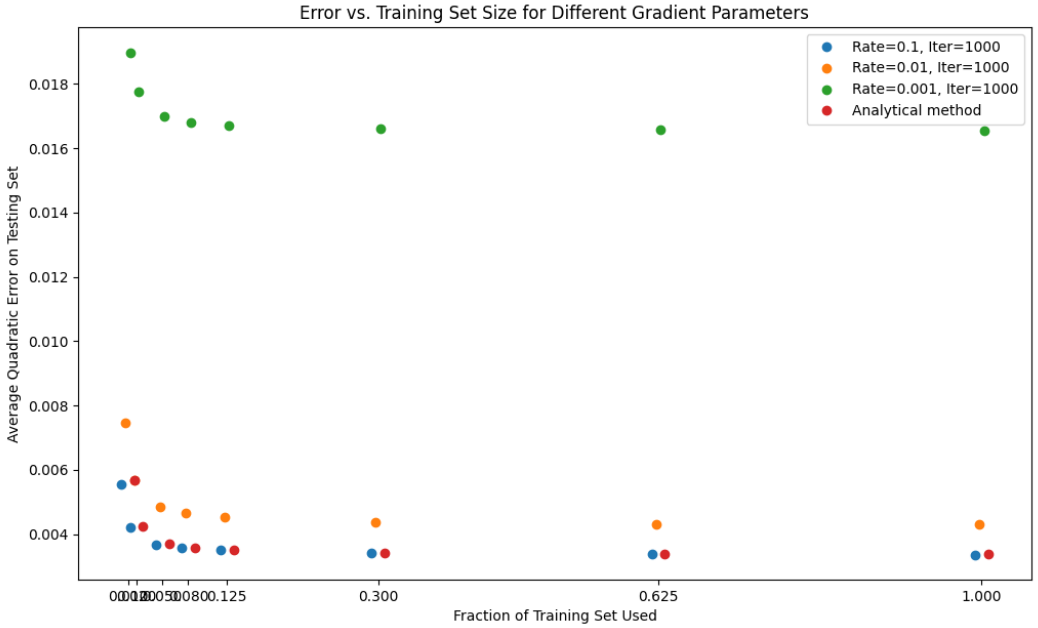
```
for j in range(GRADIENT_DESCENT_ITERATIONS):
                y_pred = X @ theta
                y_diff = [y_pred[k] - y[k] for k in range(len(y))]
                gradient = (2 / len(y)) * X.T @ y_diff
                theta = theta - GRADIENT_DESCENT_RATE * gradient
```

Rysunek 4: Schemat metody gradientu

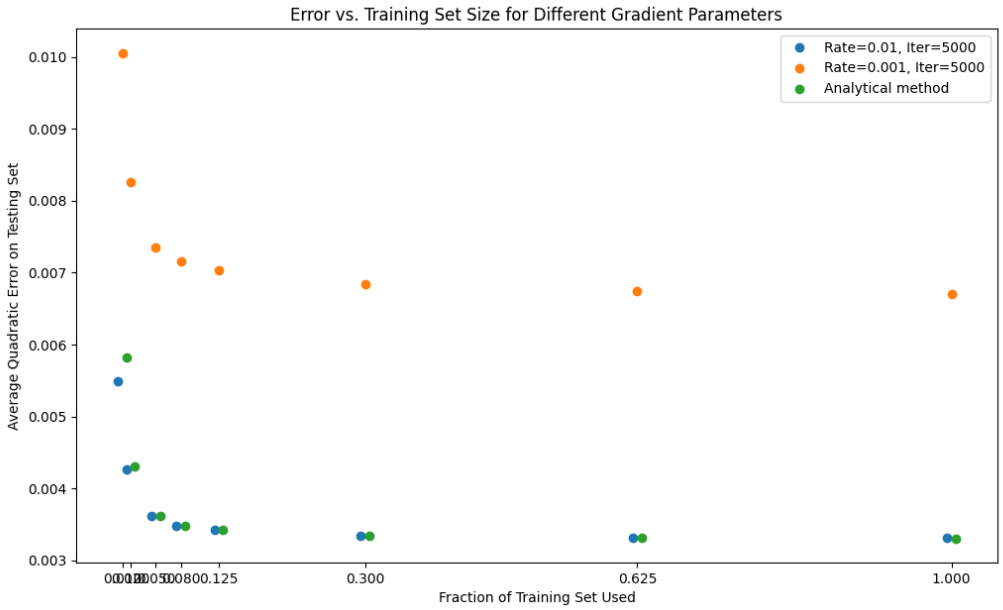
Wyniki eksperymentrów z parametrami przedstawiają wykresy:



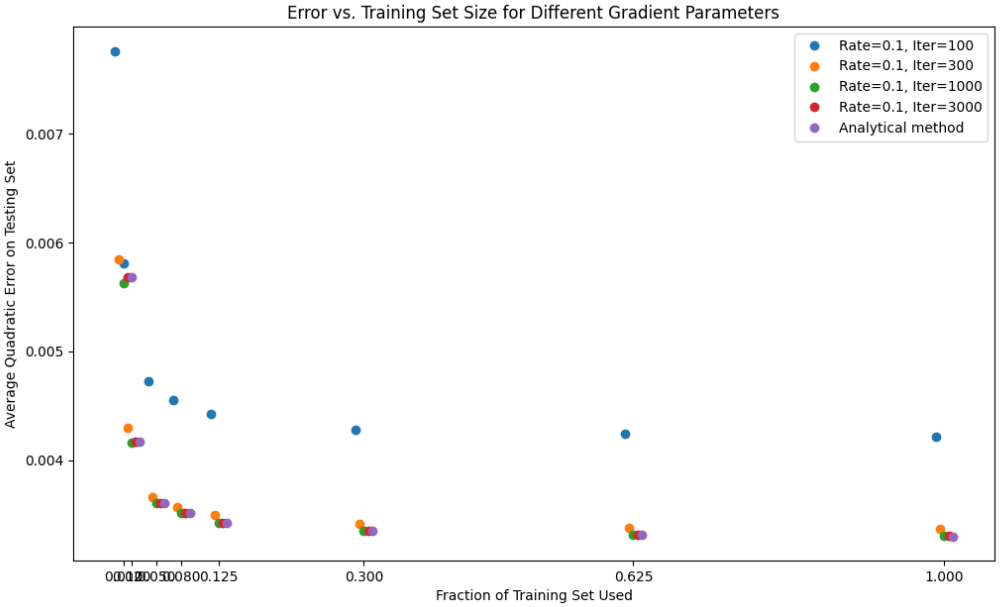

Rysunek 5: Zbyt duży współczynnik może w niektórych sytuacjach prowadzić do rozbieżności




Rysunek 6: Zbyt mały współczynnik wymaga zbyt wielu iteracji metody gradientowej




Rysunek 7: Większa liczba iteracji dla małych wskaźników




Rysunek 8: Zmiana liczby iteracji dla spadku 0.1

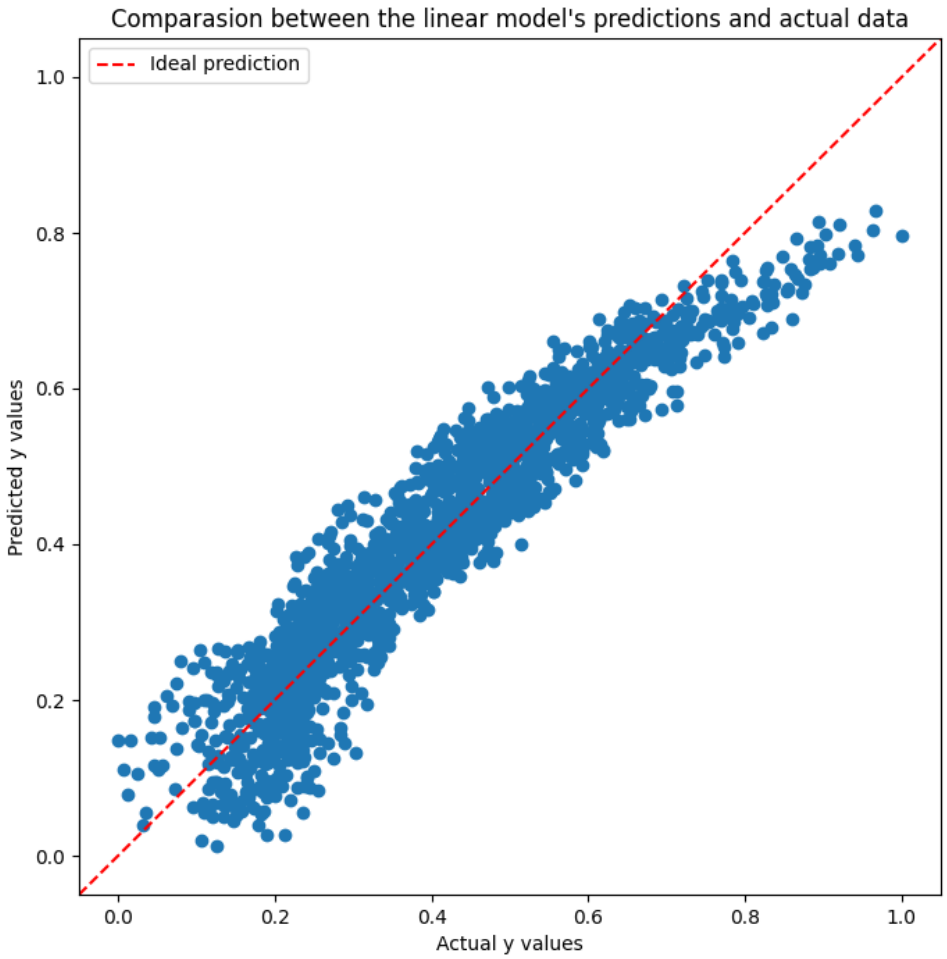
Widzimy, że współczynnik 0.1 daje bezpieczeństwo zbieżności metody gradientowej, a jednocześnie daje wyniki zbliżone do metody analitycznej przy rozsądnej liczbie iteracji.

# 3 Funkcje bazowe

Przyjrzyjmy się temu, jak z przewidywaniem wartości zmiennej objaśnianej na wszystkich danych poradził sobie model podstawowy. Wśród modeli, w których dopuściliśmy wykorzystanie całego zbioru treningowego, średnia wartość obliczonego wektora $\theta$ miała postać:

$$\theta^T = [0.151, -0.018, -0.009, 0.529, 0.177, 0.002, -0.010]$$

Poniższy wykres porównuje wartości zmiennej objaśnianej obliczonej z otrzymanego wektora zależności liniowej, oraz wartości faktycznej odczytanej z dostarczonych danych:




Rysunek 9: Skuteczność modelu liniowego

Mamy podejrzenia, że zależność może nie być liniowa. Widzimy, że model ma największą skuteczność mniej więcej w połowie możliwych wartości y, natomiast największe błędy obserwujemy przy skrajnych wartościach zmiennej objaśnianej. Wypróbujemy modele posługujące się innymi funkcjami bazowymi.

## 3.1 Funkcje wielomianowe

Porównamy modele zakładające, że dane mogą być skorelowane wielomianowo, tzn. zmienna objaśniana może być obliczona ze wzoru:

$$y = \sum_{i=1}^{7} \theta_i x_i^k$$

dla $k > 1$.
Sposób obliczania gradientu jest bardzo podobny, chcąc zminimalizować $J(\theta) = \frac{1}{m}(X^k\theta - y)(X^k\theta - y)$ pochodna po wektorze $\theta$ ma postać:

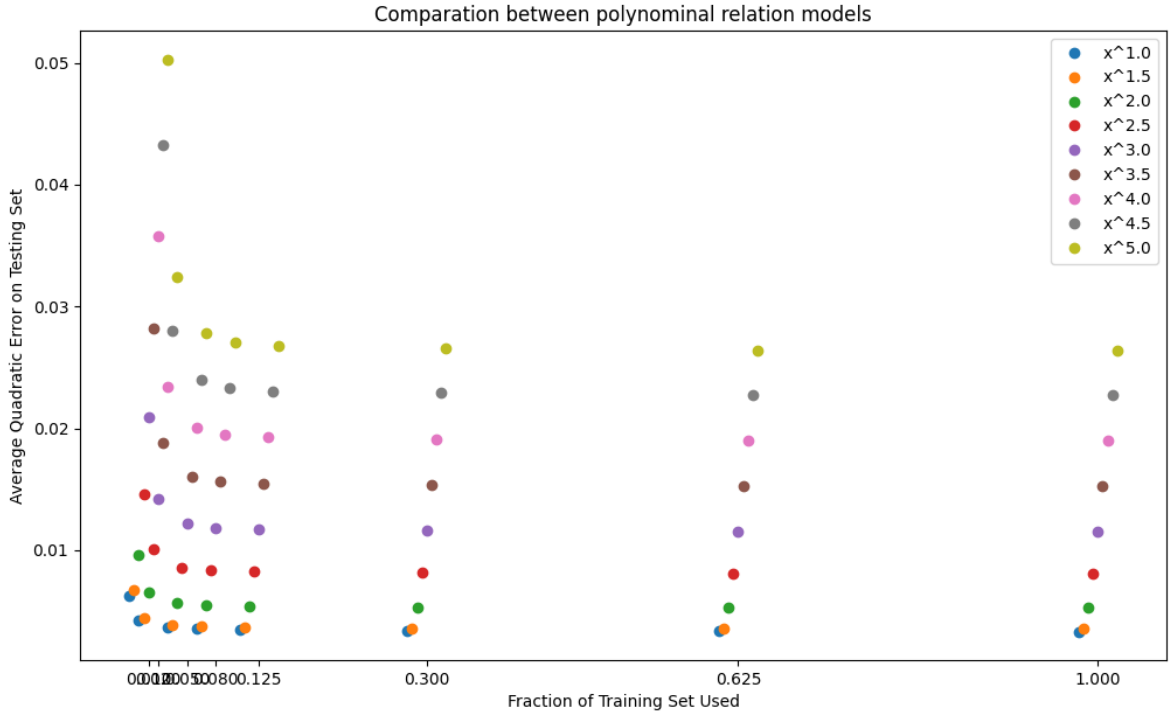
$$\frac{dJ(\theta)}{d\theta} = \frac{2}{m}(X^k)^T(X^k\theta - y)$$

gdzie zapis $X^k$ oznacza macierz, w której każdy element z $X$ podnosimy do potęgi $k$.
Do obliczania wektora theta metodą gradientową posłuży zmodyfikowany kod:

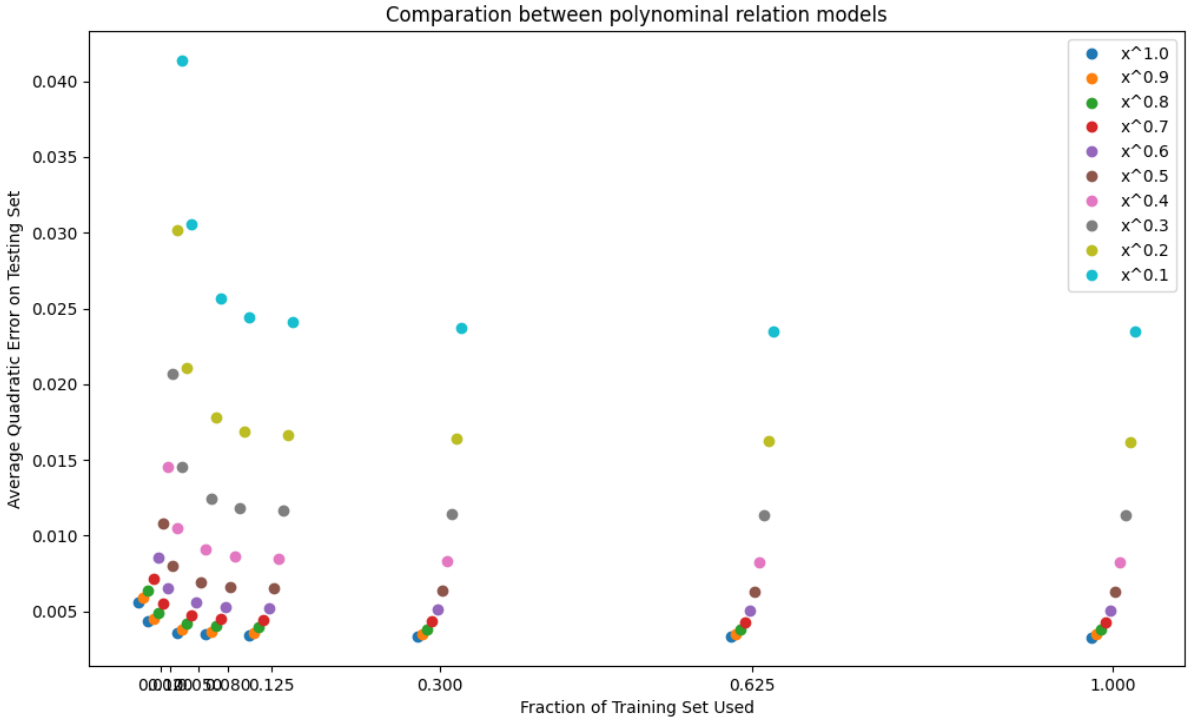
```
X, y = ds.CreateMatrices(trainingSetSubset)
theta = [0.0 for _ in range(7)]
X_pow = [[pow(i,gradientParams[2]) for i in X[j]] for j in range(len(X))]
X_pow = np.array(X_pow)

if GRADIENT_DESCENT_ITERATIONS != 0:
    #Gradient method
    for j in range(GRADIENT_DESCENT_ITERATIONS):

        #Modified gradient function
        y_pred = X_pow @ theta
        y_diff = [y_pred[k] - y[k] for k in range(len(y))]
        gradient = (2 / len(y)) * X_pow.T @ y_diff
        theta = theta - GRADIENT_DESCENT_RATE * gradient
```

Rysunek 10: Metoda gradientu dla relacji wielomianowej

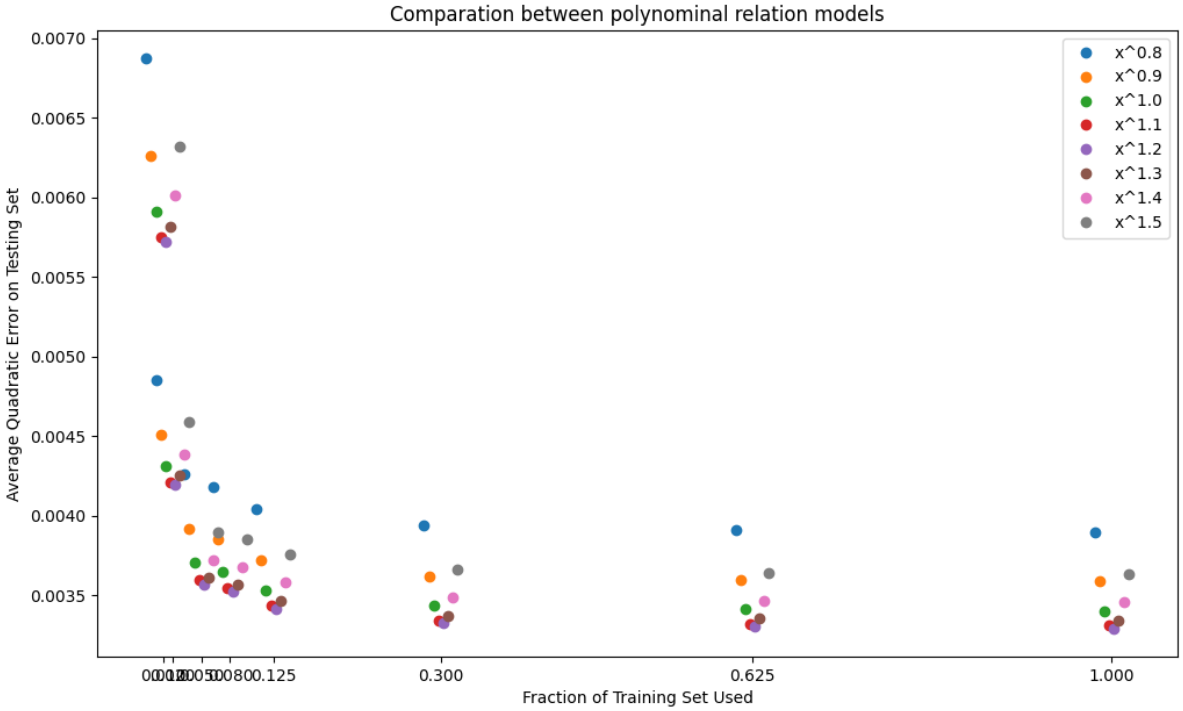
Sprawdzamy jakie wyniki uzyskują algorytmy dla różnych wartości parametru $k$:
Warto również zaznaczyć, że możemy testować ten model dla $k \in [0, 1]$, ponieważ na początku przeskalowaliśmy wszystkie dane tak, żeby znalazły się w zakresie $[0, 1]$. Nie musimy się więc martwić o wyciąganie pierwiastków z liczb ujemnych.



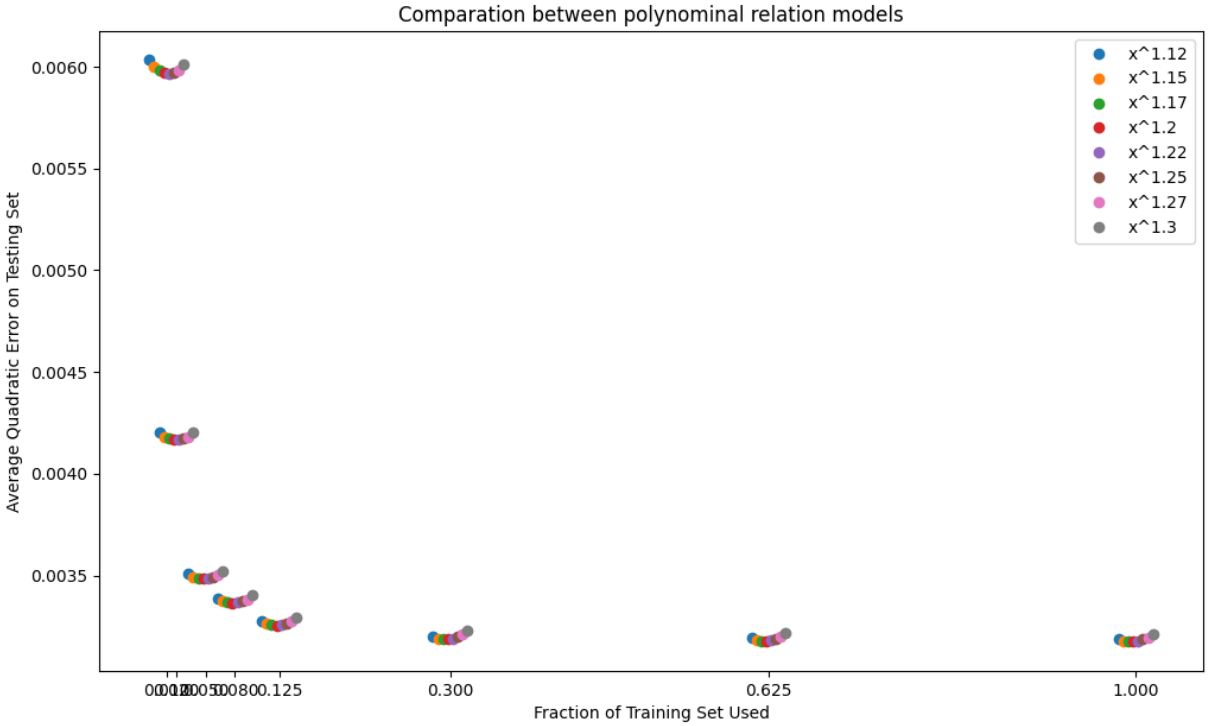

Rysunek 11: Duże wartości parametru $k$




Rysunek 12: Małe wartości parametru $k$




Rysunek 13: Dalsze porównania w przedziale $[0.8, 1.5]$

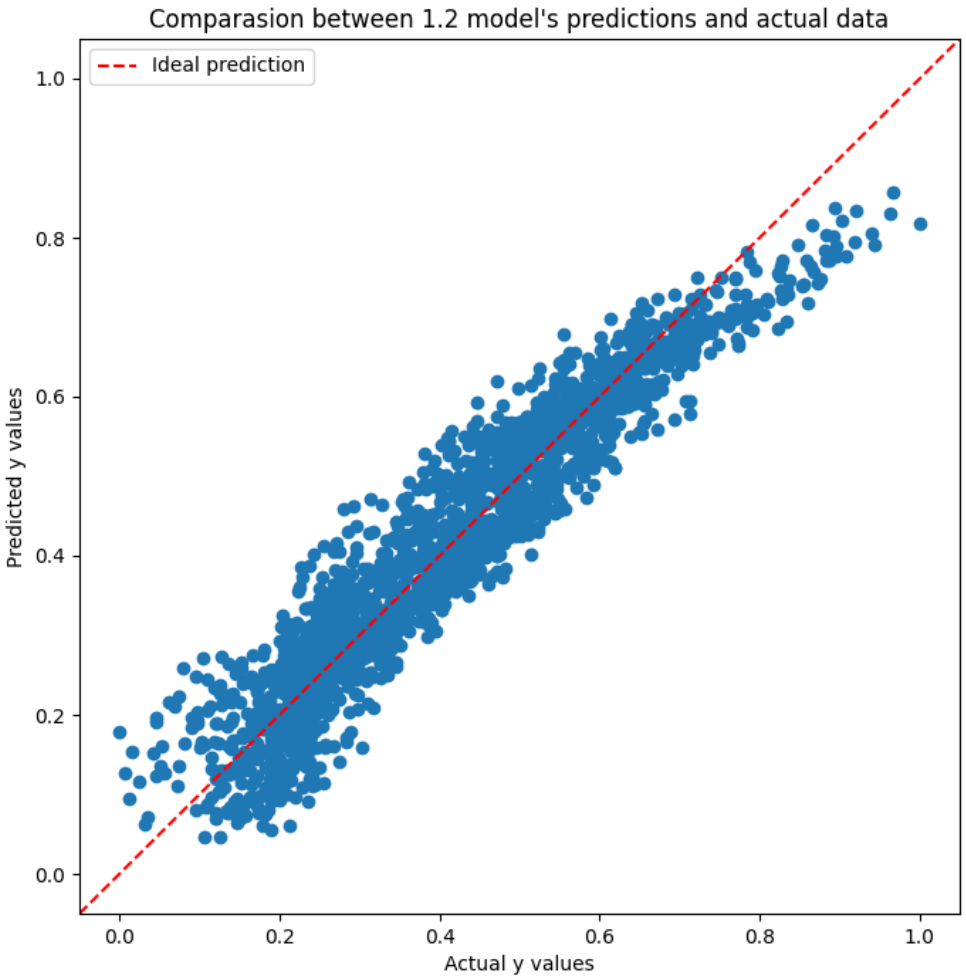



Rysunek 14: Dalsze porównania w przedziale $[1.12, 1.3]$

Pierwsze 2 wykresy przedstawiają średnie błędy uzyskane przez modele badające zależność wielomianową względem danego parametru $k$. Jak widać, zarówno zwiększanie jak i zmniejszanie tego parametru o zbyt dużą wartość prowadzi do o wiele większych błędów od tych uzyskanych przy zależności liniowej.

Modele z parametrami 0.9 oraz 1.5 zachowywały się jednak dosyć dobrze, dawały zbliżone wyniki. Po dalszych poszukiwaniach w zawężonych zakresach można zaobserwować lekką poprawę wyników. Uśredniając wyniki z wielu niezależnych prób, najlepiej radziły sobie modele obliczające zależność dla parametru $k = 1.2$.

Nie zaobserwowaliśmy jednak znaczącej poprawy względem modelu liniowego, wykres porównujący przewidywane wartości z rzeczywistymi nadal wygląda bardzo podobnie, a średni błąd oscyluje w granicach 0.0033.




Rysunek 15: Skuteczność modelu z parametrem $k = 1.2$

## 3.2 Iloczyny cech

Spróbujemy sprawdzić, czy wśród danych występują zależności między poszczególnymi cechami, których uwzględnienie w przewidywaniu wartości zmiennej objaśnianej mogłoby znacząco poprawić wyniki. Wypróbujemy model, w którym zachodzi zależność:

$$y = \sum_{i,j \in [1,7]} \theta_{i,j} x_i x_j$$

Chcemy więc zminimalizować następującą funkcję straty na zbiorze treningowym:

$$J(\theta) = \frac{1}{m}\sum_{k=1}^{m}(\sum_{i,j\in[7],i\leqslant j}\theta_{i,j}x_{k,i}x_{k,j} - y_k)^2$$

Gradient obliczamy w następujący sposób:

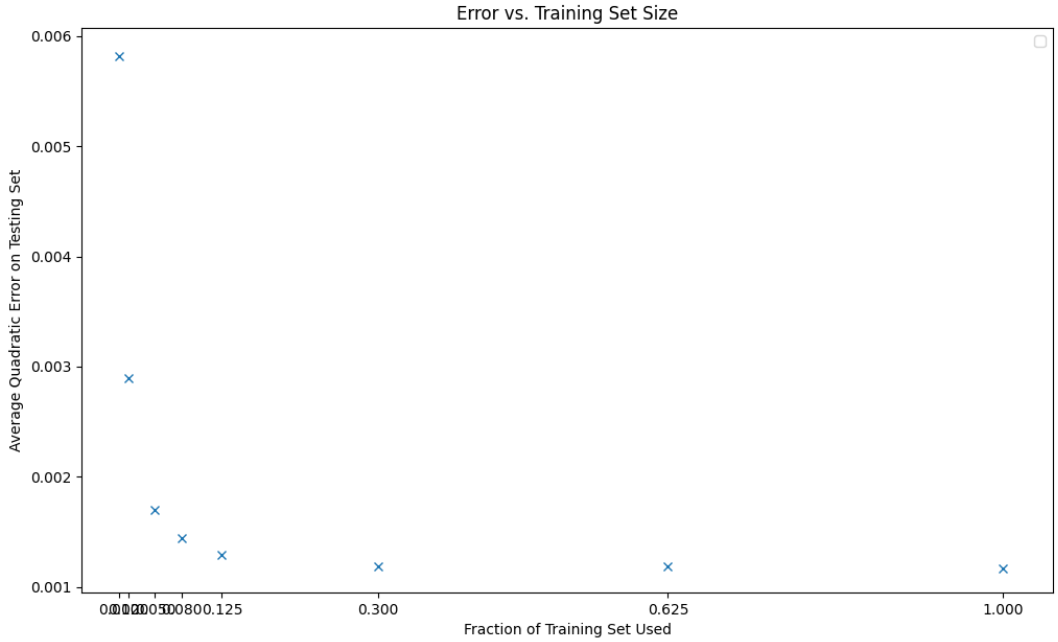
$$\frac{dJ(\theta)}{d\theta_{i,j}} = \frac{2}{m}\sum_{k=1}^{m}((\sum_{i,j\in[7],i\leqslant j}\theta_{i,j}x_{k,i}x_{k,j} - y_k)x_{k,i}x_{k,j})$$

Do obliczania parametru $\theta$ posłuży następujący kod:

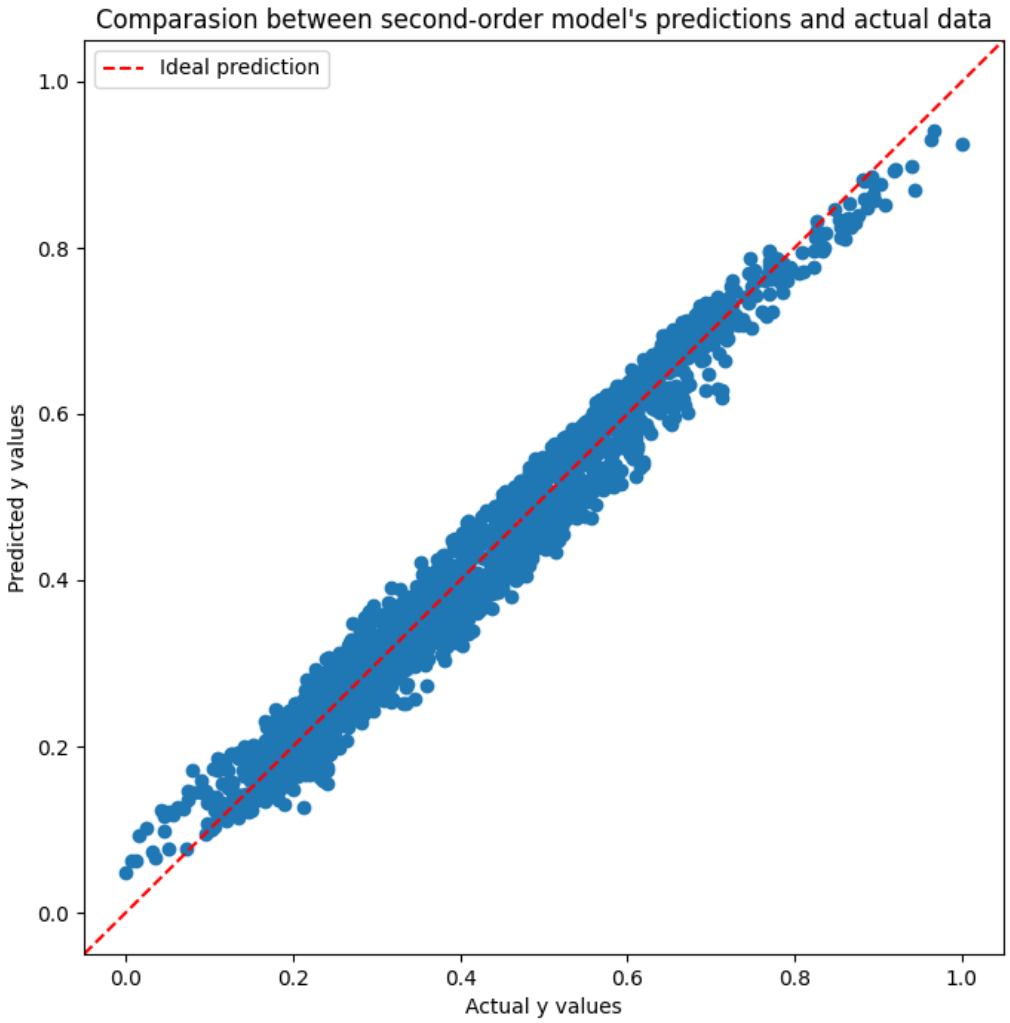
```
X, y = ds.CreateMatrices(trainingSetSubset)
                theta = [[0.0 for _ in range(7)] for _ in range(7)]

                if GRADIENT_DESCENT_ITERATIONS != 0:
                    #Gradient method
                    for j in range(GRADIENT_DESCENT_ITERATIONS):

                        #
                        y_pred = []
                        m = len(X)

                        for k in range(m):
                            newYPred = 0
                            for b in range(7):
                                for a in range(0, b+1):
                                    newYPred += theta[a][b] * X[k][a] * X[k][b]
                            y_pred.append(newYPred)

                        y_diff = [y_pred[k] - y[k] for k in range(len(y))]

                        gradient = [[0.0 for _ in range(7)] for _ in range(7)]

                        for b in range(7):
                            for a in range(0, b+1):
                                #Calculate Grad_a_b
                                for k in range(m):
                                    gradient[a][b] += y_diff[k] * X[k][a] * X[k][b]
                                gradient[a][b] *= (2/m)
                                gradient[a][b] = float(gradient[a][b])

                        theta = [[theta[c][d] - GRADIENT_DESCENT_RATE * gradient[c][d] for d in range(7)] for c in range(7)]
```

Rysunek 16: Metoda gradientu dla iloczynów różnych cech

Wyniki przedstawione są na wykresach na kolejnej stronie. Jak widać, udało się znacznie zmniejszyć otrzymywane błędy, model dużo lepiej dopasowuje się do faktycznych danych.




Rysunek 17: Krzywa uczenia modelu z iloczynami cech



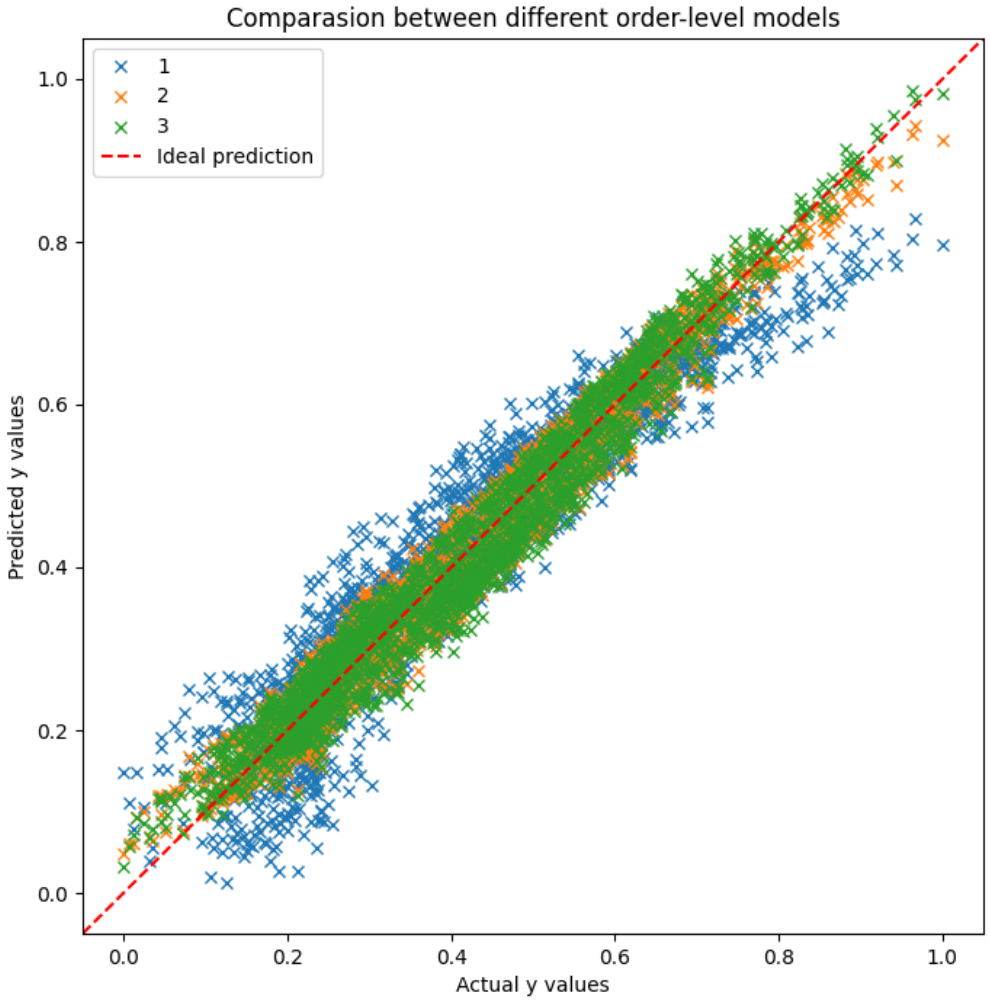

Rysunek 18: Skuteczność modelu z iloczynami cech

### 3.3 Wyższe rzędy iloczynów

Wypróbowałem również model, w którym $y$ może zależeć od wszystkich możliwych kombinacji trzech cech spośród zbioru. Szukał on $\theta$ najlepiej objaśniającego zależność:

$$y = \sum_{i,j,k \in [1,7], i \leqslant j \leqslant k} \theta_{i,j,k} x_i x_j x_k$$

Porównanie skuteczności modeli wszystkich trzech rzędów przedstawia wykres:




Rysunek 19: Porównanie skuteczności modeli

Co ciekawe, model trzeciego rzędu dużo lepiej dopasował się do próbek, w których wartość zmiennej objaśnianej była bliska wartości maksymalnej. Jednak ostatecznie średni błąd był nieco wyższy niż ten uzyskany z modelu drugiego rzędu. MSE dla tych modeli wynosiły kolejno:

- Model 1 rzędu: (0.003365, 29422)
- Model 2 rzędu: (0.001235, 10798)
- Model 3 rzędu: (0.001285, 11235)

gdzie pierwsza wartość została obliczona na podstawie wartości przeskalowanych danych, a drugą obliczamy korzystając z faktu, że zmienne $y$ pierwotnie znajdowały się w przedziale $[-238.06, 2718.9]$, więc aby otrzymać błąd na pierwotnych zmiennych wystarczy pomnożyć pierwszy przez $(2718.9 - (-238.06))^2$.

# 4 Regularyzacja

Do najlepiej sprawujących się modeli dodamy parametr regularyzacji. W tym celu do funkcji kosztu dodamy parametr mający na celu zapobieganie zbyt dużym wartościom wektora $\theta$ mogącym prowadzić do przeuczenia na zbiorze testowym

Dla regularyzacji $l_2$ definiujemy nową funkcję kosztu:

$$J_{regl_2}(\theta) = J(\theta) + \lambda||\theta||_2$$

Pochodna po danej składowej wektora $\theta$ zmienia się więc następująco:

$$\frac{dJ_{regl_2}(\theta)}{d\theta_S} = \frac{dJ(\theta)}{d\theta_S} + 2\lambda\theta_S$$

Analogicznie dla regularyzacji $l_1$:

$$J_{regl_1}(\theta) = J(\theta) + \lambda||\theta||_1$$

$$\frac{dJ_{regl_1}(\theta)}{d\theta_S} = \frac{dJ(\theta)}{d\theta_S} + \lambda sgn(\theta_S)$$

Parametr lambda będziemy wyznaczać za pomocą zbioru walidacyjnego. Zaczniemy od listy kilku potencjalnych wartości, oddalonych od siebie równo na skali logarytmicznej, żeby od początku przetestować dosyć szeroki zakres potencjalnych parametrów. Następnie po znalezieniu najlepszego wypełnimy tablicę jego okolicznymi wartościami, i powtórzymy iteracje.

Aby przyspieszyć działanie algorytmu, parametr lambda będziemy obliczać przy zmniejszonej liczbie iteracji metody gradientowej. Mimo oczywistej wynikającej z tego gorszej precyzji algorytmu, zależności pomiędzy różnymi parametrami lambda powinny pozostać nieruszone, dzięki czemu możemy niższym kosztem wyznaczyć ten parametr. Oczywiście, przy trenowaniu modelu na już wyznaczonym parametrze wykorzystamy pełną liczbę iteracji metody gradientowej.

Dla kilku niezależnych przebiegów algorytmu uczenia się zapiszemy obliczone wektory $\theta$, następnie obliczymy ich średnią, i porównamy, jak sprawdzają się w porównaniu z tymi obliczonymi przez modele nie uwzględniające parametru regularyzacji.

Błędy uzyskane przy zastosowaniu regularyzacji dla modelu liniowego prezentują się następująco:

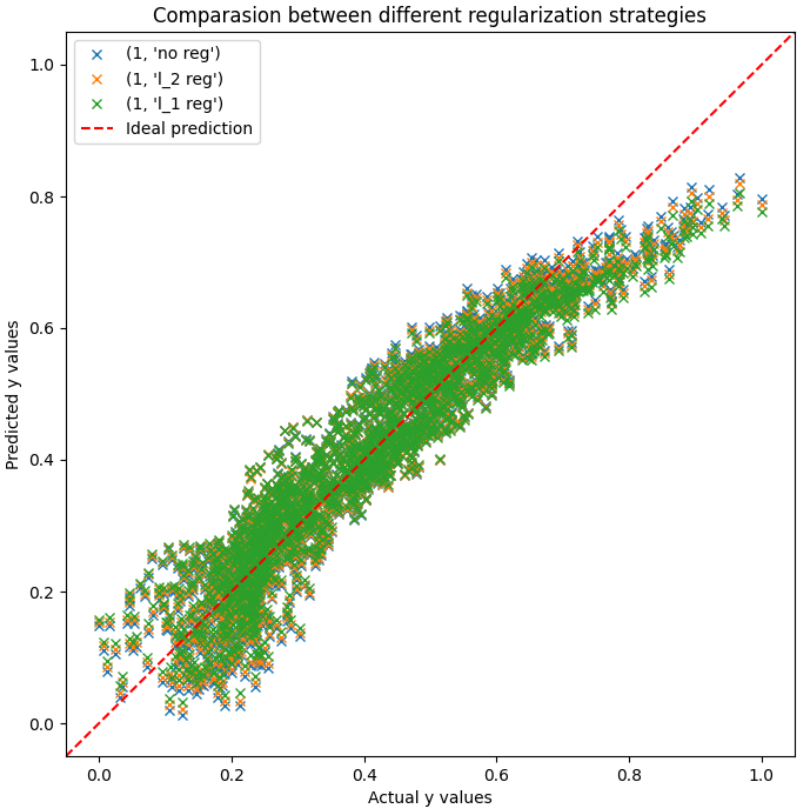
- Model 1 rzędu bez regularyzacji: (0.003365, 29422)
- Model 1 rzędu z regularyzacją $l_2$: (0.003324, 29063)
- Model 1 rzędu z regularyzacją $l_1$: (0.003294, 28801)

A tak wyglądają błędy dla modelu rzędu 2, czyli uzwględniającego relacje między parametrami:

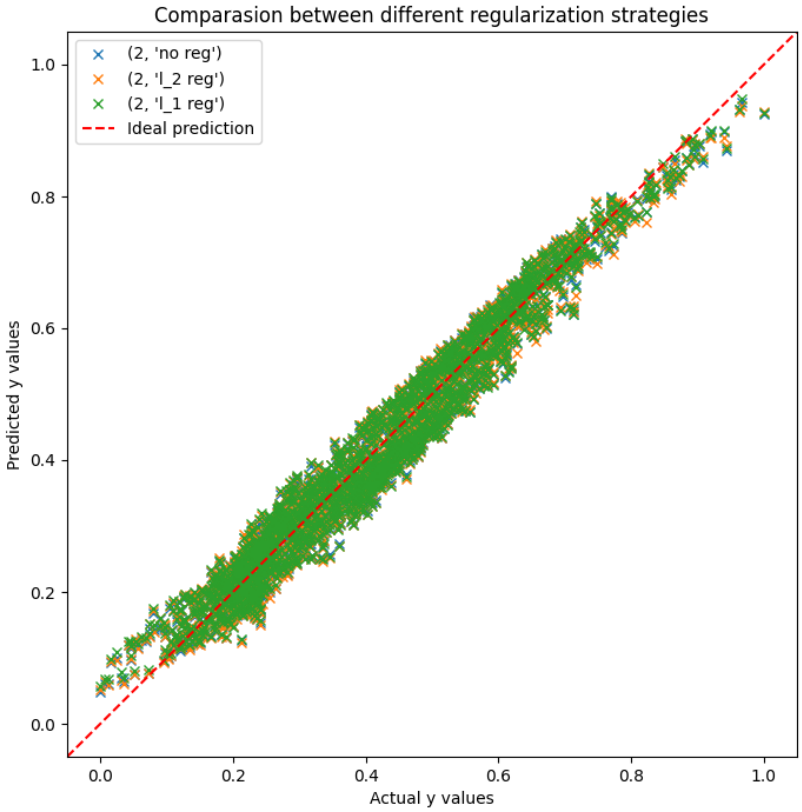
- Model 2 rzędu bez regularyzacji: (0.001235, 10798)
- Model 2 rzędu z regularyzacją $l_2$: (0.001324, 11576)
- Model 2 rzędu z regularyzacją $l_1$: (0.001350, 11803)

Predykcje zestawione z faktycznymi wartościami przedstawione na wykresach. Jak widać, regularyzacja daje nieznaczną poprawę. W przypadku drugiego modelu, uzyskaliśmy gorszy wynik stosując regularyzację niż ignorując ją. Ups.

Wnioskujemy, że w badanych modelach nie nastąpiło zjawisko overfittingu, modele nie przeuczały się na danych treningowych, przez co wyniki dla zbioru testowego były dobre bez konieczności stosowania regularyzacji.




Rysunek 20: Porównanie skuteczności modeli 1 rzędu w zależności od regularyzacji




Rysunek 21: Porównanie skuteczności modeli 2 rzędu w zależności od regularyzacji