COSC 2753 | Machine Learning

Week 4 Lab Exercises: **Polynominal Regression and k-fold Cross Validation**

Introduction

Last weeks we learned how to read the data, do exploratory data analysis (EDA), split data, feed the data to a learning algorithm.

The lab assumes that you have completed `Week 02 lab: Reading data & Exploratory Data Analysis (EDA)` and `Week 03 lab: Training a Regression Model`. If you haven't yet, please do so before attempting this lab.

In this lab, we will practise performing k-fold cross validation and use it to find the best regularisation parameter for a lasso polynomial regression model.

> ⚠ **Warning: Starting this week, we will progressively provide less code, and would like you to use previous labs and what you know to perform the tasks. This will help you to become proficient at this.**

The lab can be executed on either your own machine (with anaconda installation) or computer lab.

- Please refer canvas for instructions on installing anaconda python

Objective

- Continue to familiarise with Python and other ML packages
- Practice polynomial regression
- Practice performing k-fold cross validation
- Use validation set to find best regularisation parameter

Dataset

We contineously examine two regression based datasets in this lab. The first one is to do with house prices, some factors associated with the prices and trying to predict house prices. The second dataset is predicting the amount of share bikes hired every day in Washington D.C., USA, based on time of the year, day of the week and weather factors. These datasets are available in `housing.data.csv` and `bikeShareDay.csv` in the code repository.

First, ensure the two data files are located within the Jupyter workspace.

- If you are on the local machine copy the two data data directories (`BostonHousingPrice`,`Bike-Sharing-Dataset`) to your current folder.

In this course we mostly use datasets that are collected by a third party. If you are interested in collecting your own data for your project, some useful information can be found at: [Introduction to Constructing Your Dataset]

Problem Formulation

The first step in developing a model is to formulate the problem in a way that we can apply machine learning. To reiterate, the `task` in the Boston house price dataset is to predict the house price (`MEDV`), using some attributes of the house and neighbourhood.

� Observe the data and see if there is a pattern that would allow us to predict the house price using the attributes given? You can use the observations from the EDA for this.

� What category does the task belong to?

> ✔ **Task category:**
>
> - supervised, univariate/multivariate regression
>
> - We should use the insights gained from observing the data (EDA) in selecting the performance measure. e.g. are there outliers in target?

Load dataset

Start a new Jupyter notebook session. Load the dataset 'housing.data.csv'in bostonHouseFrame

```
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np

## TODO
bostonHouseFrame = pd.read_csv("housing.data.csv", delimiter="\s+")
print(bostonHouseFrame)
```

```
        CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD    TAX  \
0    0.00632  18.0   2.31     0  0.538  6.575  65.2  4.0900    1  296.0
1    0.02731   0.0   7.07     0  0.469  6.421  78.9  4.9671    2  242.0
2    0.02729   0.0   7.07     0  0.469  7.185  61.1  4.9671    2  242.0
3    0.03237   0.0   2.18     0  0.458  6.998  45.8  6.0622    3  222.0
4    0.06905   0.0   2.18     0  0.458  7.147  54.2  6.0622    3  222.0
..       ...   ...    ...   ...    ...    ...   ...     ...  ...    ...
501  0.06263   0.0  11.93     0  0.573  6.593  69.1  2.4786    1  273.0
502  0.04527   0.0  11.93     0  0.573  6.120  76.7  2.2875    1  273.0
503  0.06076   0.0  11.93     0  0.573  6.976  91.0  2.1675    1  273.0
504  0.10959   0.0  11.93     0  0.573  6.794  89.3  2.3889    1  273.0
505  0.04741   0.0  11.93     0  0.573  6.030  80.8  2.5050    1  273.0

     PTRATIO       B  LSTAT  MEDV
0       15.3  396.90   4.98  24.0
1       17.8  396.90   9.14  21.6
2       17.8  392.83   4.03  34.7
3       18.7  394.63   2.94  33.4
4       18.7  396.90   5.33  36.2
..       ...     ...    ...   ...
501     21.0  391.99   9.67  22.4
502     21.0  396.90   9.08  20.6
503     21.0  396.90   5.64  23.9
504     21.0  393.45   6.48  22.0
```

```
505     21.0  396.90   7.88  11.9

[506 rows x 14 columns]
```

Univariate Regression

We will first study how to do univariate regression.

If you recall from the last lab, we found that possibly the ‘RM’ (number of rooms) and ‘LSTAT’ (unsure) variables seem to have a linear relationship with the house price (‘MEDV’). Hence, we will try these variables as the independent variable to predict the house price, i.e., the dependent variable.

- Create an independent variable that is just based on the ‘RM’ column and a dependent variable of ‘MEDV’.
- Assign the values of the 'RM' column to a variable named 'house_uniRM_x' and assign the values of the 'MEDV' column to a variable named 'house_y'.

� There are at least three ways to create the house_uniRM_x and house_y variables based on those columns. Do some quick online research to slice the bostonHousePrice dataset and extract the two columns, by using the following methods:

- (1) using square brackets []
- (2) using the pandas function .loc
- (3) using the pandas function .iloc

```
#print(bostonHouseFrame)
#house_uniRM_x = bostonHouseFrame.loc[:,'RM']
#house_uniRM_x= house_uniRM_x.values.reshape(-1,1)
house_uniRM_x = bostonHouseFrame[['RM']]
house_y = bostonHouseFrame['MEDV']
```

```
print(house_y)
print(house_uniRM_x.shape)
```

```
0      24.0
1      21.6
2      34.7
3      33.4
4      36.2
       ...
501    22.4
502    20.6
503    23.9
504    22.0
505    11.9
Name: MEDV, Length: 506, dtype: float64
(506, 1)
```

Hold-out Validation

As we have discussed in the lecture, in supervised learning we are interested in learning a model using our dataset that can predict the target value for unseen data (Not in the training set). This is called **generalization**. How can we test if the model we developed with our training data would generalize? One approach we can use is to **hold some data from the training process - hold-out validation** (Hypothetical unseen data). This hold out data subset (split) is called the `"Test set"` and the remaining data is called the `”Training set"`. The training set may be further subdivided, but more on this later in the regularization lecture. We can use the "Test set" at the end of the development phase to test our model and see if it generalizes.

- **Training set:** Is applied to train, or fit, your model. For example, you use the training set to find the optimal weights, or coefficients, for linear regression, logistic regression, or neural networks.
- **Test set:** Needed for an unbiased evaluation of the final model.

> ⚠ **Warning: The test set should be independent and identically distributed with respect to the training data**
>
> - Should make sure that there is no leakage between the two sets (overlapped train and test instances). This will give unrealistically high performance metric values for your model. e.g. In house price prediction, there may be a house that was sold multiple times and, you might include some instances of this house in the train set and some in the test set. This will result in data leakage.
> - There should be no underlying differences between the two distributions. In other words the characteristics of the test set should not be different to that of the train set. For example all the houses sold in winter in train set and all the houses sold on summer in another set (generally, there is a difference in house prices sold in winter vs summer).
> - More on this in the lectures.

> ⚠ **Warning: The test data should NOT be used for any aspect of the model development process (training).**
> This includes hyper parameter tuning and model selection (a separate validation set should be used for them).

Next, we want to create some data to train the model, and some other data to evaluate how good the model is. We may be tempted to use 100% of the data for training, then select a subset for evaluation (say 20% of the data). However, we will find out later that this isn’t a good idea generally, as the data we use to test is the same we use to train, and likely to cause overfitting and what is called bias. We will discuss this more in a later week, but for now, just note that it isn’t a good idea to do so.

Instead, we will split the data into a training set, which we use to fit the model/hypothesis, and a testing set, which we will use to evaluate the performance of the fitted model/hypothesis. There should be no overlap between the two sets. How we achieve this is typically randomly split the data, say 80% for training and 20% for testing. We can do this ourselves, but like many machine learning functionality these days, they are already implemented, and of course, they are available in the libraries we have imported.

The scikit-learn Python machine learning library provides an implementation of the train-test splitting via function `train_test_split()`. Lets use this to randomly split our data to 80% train set and 20% test set.

```
# create testing and training data for RM variable
from sklearn.model_selection import train_test_split
trainX, testX, trainY, testY = train_test_split(house_uniRM_x, house_y, test_size=0.2,shuffle=True)
print(trainX.shape)
print(testX.shape)
print(trainY.shape)
print(testY.shape)
```

```
(404, 1)
(102, 1)
(404,)
(102,)
```

The function that does the work is ‘train_test_split()’, part of sklearn.model_selection package. It essentially does what we desire, with the first two arguments to it are the X and Y variable datasets respectively (they must be of the same number of rows/instances), and a test_size parameter which specifies how big the test dataset is (in this case, 20% of the data, which is a typical setting for this). This function returns ‘trainX’ (training data of the X variable), ‘testX’ (testing data of X), ‘trainY’ (training data of Y) and ‘testY’ (testing data of Y). The last four statements just print out the size of the resulting training and testing datasets to show you that the training (test) datasets contain the same number of rows. We will use trainX and trainY to train the linear regression model, then textX and testY for testing and evaluation.

We are now ready to construct the linear regression model/hypothesis and fit the theta parameters.

Polynomial Regression

Now, implement two linear regression models:

- One with order 1 polynomial (i.e., the simplest model)
- And the other with order 4 polynomial. To implement the latter, please have a look at this documentation: http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html

Order 1 Polynomial Regression

Now, implement order 1 polynomial - linear regression models

```
from sklearn import linear_model
linReg = linear_model.LinearRegression()
```

This constructs the linear regression model object, assigned to variable ‘linReg’. We then fit the training data to the model:

```
linReg.fit(trainX, trainY)
```

```
▾ LinearRegression
LinearRegression()
```

linReg.fit() fits the X and Y training data and optimises the parameters to minimise the loss function. It might not exactly use gradient descent, but the ideas are similar and as stated in lectures, gradient descent as a general optimisation is the crux of many optimisation and parameter fitting algorithms.

Let's have a look at what the parameters look like:

```
print(linReg.intercept_)
print(linReg.coef_)
```

```
-36.10374278242093
[9.3246616]
```

linReg.intercept_ is the y-intercept, or essentially the theta parameter.

linReg.coef_ is the slope of the univariate linear regression line or the theta_one variable.

With a model/hypothesis, we can now do prediction! We use the testing data for that:

```
pred_uniRM_y = linReg.predict(testX)
```

Okay, we now have predictions, but how did we go? What we want to do is to compare the predicted value from our model (given testX) with the actual Y values (testY). We can estimate the error (using the mean squared error loss function we been discussing in lectures for fitting the parameters), as follows:

```
from sklearn.metrics import mean_squared_error
print('Mean squared error ', mean_squared_error(testY, pred_uniRM_y))
```

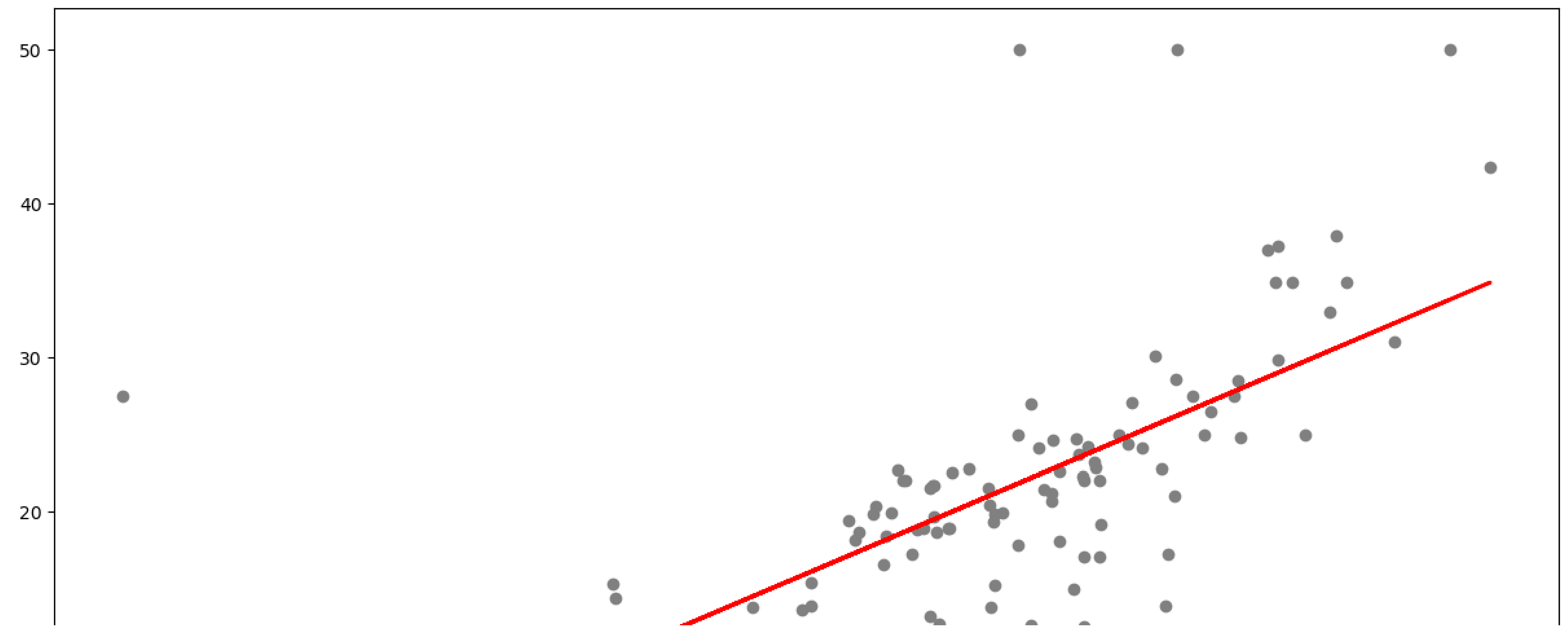
```
Mean squared error  46.67135913580404
```

What is the mean squared error value you obtained?

In addition, it is useful for regression to plot the testing data against the model/hypothesis, which is a line in our univariable linear regression. Type in the following:

```
plt.figure(figsize=(15,9))
plt.scatter(testX, testY, color='grey')
plt.plot(testX, pred_uniRM_y, color='red', linewidth=2)
```

```
[<matplotlib.lines.Line2D at 0x1e138ae1610>]
```



This uses our reliable plotting package matplotlib and pyplot to first, produce a scatterplot of our testing data (X,Y) pairs, then draw a blue line for our model/hypothesis. What information does the plot visualise?

Another interesting visualisation and based on the same information is the residual plot, which shows what is the difference in predicted and actual values, across different X values, for the testing data;

Order 4 Polynomial Regression

To implement order 4 Polynomial Regression, please have a look at this documentation: http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html

First, import PolynomialFeatures

```
from sklearn.preprocessing import PolynomialFeatures
```

STEP #1: Determining the degree of the polynomial

```
poly_feature=PolynomialFeatures(degree=4)
```

"degree" sets the degree of our polynomial function. degree=4 means that we want to work with a 4th degree polynomial! RECALL what is the formulation of 4th degree polynominal???

STEP #2: Creating the new features

There's only one method – fit_transform() – but in fact it's an amalgam of two separate methods: fit() and transform(). fit_transform() is a shortcut for using both at the same time, because they're often used together.

Since I want you to understand what's happening under the hood, I'll show them to you separately.

With fit() we basically just declare what feature we want to transform:

```
X_poly=poly_feature.fit_transform(trainX)
```

There's only one method – fit_transform() – but in fact it's an amalgam of two separate methods: fit() and transform(). fit_transform() is a shortcut for using both at the same time, because they're often used together.

Since I want you to understand what's happening under the hood, we try them separately.

With fit() we basically just declare what feature we want to transform:

```
poly_feature.fit(trainX)
```

```
▾ PolynomialFeatures
PolynomialFeatures(degree=4)
```

transform() performs the actual transformation:

```
poly_feature.transform(trainX)
```

```
array([[1.00000000e+00, 5.92600000e+00, 3.51174760e+01, 2.08106163e+02,
        1.23323712e+03],
       [1.00000000e+00, 6.40400000e+00, 4.10112160e+01, 2.62635827e+02,
        1.68191984e+03],
       [1.00000000e+00, 5.98100000e+00, 3.57723610e+01, 2.13954491e+02,
        1.27966181e+03],
       ...,
       [1.00000000e+00, 6.09500000e+00, 3.71490250e+01, 2.26423307e+02,
        1.38005006e+03],
       [1.00000000e+00, 6.81200000e+00, 4.64033440e+01, 3.16099579e+02,
        2.15327033e+03],
       [1.00000000e+00, 5.40400000e+00, 2.92032160e+01, 1.57814179e+02,
        8.52827825e+02]])
```

The provided Python code is a method named `transform` from a class that generates polynomial and interaction features. This method is typically part of classes like `PolynomialFeatures` in the `sklearn.preprocessing` module. The purpose of this method is to transform the input data into a new dataset containing all polynomial combinations of the features with degree less than or equal to the specified degree.

The `transform` method takes as input a 2D array-like or sparse matrix `X` of shape `(n_samples, n_features)`. The method first checks if the transformer is fitted using the `check_is_fitted` function. Then, it validates the input data `X` and gets its shape.

The method then checks if the input is a CSR (Compressed Sparse Row) matrix. If it is, and the degree of the polynomial is greater than 3, it converts the matrix to CSC (Compressed Sparse Column) format, transforms it, and then converts it back to CSR. If the degree is less than or equal to 3, it generates the polynomial features and stacks them together. If the input is a CSC matrix and the degree is less than 4, it converts the matrix to CSR, transforms it, and then converts it back to CSC. If the input is a sparse matrix of another type, it generates the polynomial features and stacks them together.

If the input is not a sparse matrix, it creates an empty array `XP` of shape `(n_samples, self._n_out_full)`. It then generates the polynomial features by iterating over the degrees and multiplying the appropriate features together. If the minimum degree is greater than 1, it slices the generated features to only include those of degree greater than or equal to the minimum degree.

Finally, the method returns the transformed data `XP`, which is a matrix of shape `(n_samples, NP)`, where `NP` is the number of polynomial features generated from the combination of inputs. If a sparse matrix was provided as input, it will be converted into a sparse `csr_matrix`.

What are these numbers? Well, remind that we wanted to create x^4 values from our x values. In the second column we have our values for x. In the second column we have our values for x squared, ect.

Looks familiar? Our 4th degree polynomial formula, again:

y = theta_0 + theta_1*x + theta_2*x^2 + theta_3*x^3 + theta_4*x^4

We save the result to X_poly:

```
#poly_reg.fit(X_poly,trainY)
X_poly=poly_feature.fit_transform(trainX)
print(X_poly)
```

```
[[1.00000000e+00 5.92600000e+00 3.51174760e+01 2.08106163e+02
  1.23323712e+03]
 [1.00000000e+00 6.40400000e+00 4.10112160e+01 2.62635827e+02
  1.68191984e+03]
 [1.00000000e+00 5.98100000e+00 3.57723610e+01 2.13954491e+02
  1.27966181e+03]
 ...
 [1.00000000e+00 6.09500000e+00 3.71490250e+01 2.26423307e+02
  1.38005006e+03]
 [1.00000000e+00 6.81200000e+00 4.64033440e+01 3.16099579e+02
  2.15327033e+03]
 [1.00000000e+00 5.40400000e+00 2.92032160e+01 1.57814179e+02
  8.52827825e+02]]
```

STEP #3: Creating the polynomial regression model

Now it’s time to create our machine learning model. Of course, we need to import it first:

```
from sklearn.linear_model import LinearRegression
```

Hold up a minute! 😮 Isn’t this tutorial supposed to be about polynomial regression? Why are we importing LinearRegression then?

Just think back to what you’ve read not so long ago: polynomial regression is a linear model, that’s why we import LinearRegression. 🙂

Let’s save an instance of LinearRegression to a variable:

```
polyReg = LinearRegression()
```

Then we fit our model to our data:

```
polyReg.fit(X_poly, trainY)
```

```
▾ LinearRegression
LinearRegression()
```

```
print(polyReg.intercept_)
print(polyReg.coef_)
```

```
-882.7034738860469
[   0.          644.10689909 -168.64588167   19.06688975   -0.77990963]
```

Now that our model is properly trained, we can put it to work by instructing it to predict the responses (y_predicted) based on poly_feature, and the coefficients it had estimated.

Prepare the data of test for Polynomial!!!

```
X_test_poly=poly_feature.fit_transform(testX)
print(X_test_poly)
```

```
[[1.00000000e+00 5.89800000e+00 3.47864040e+01 2.05170211e+02
  1.21009390e+03]
 [1.00000000e+00 6.33300000e+00 4.01068890e+01 2.53996928e+02
  1.60856255e+03]
 [1.00000000e+00 6.40500000e+00 4.10240250e+01 2.62758880e+02
  1.68297063e+03]
 [1.00000000e+00 7.13500000e+00 5.09082250e+01 3.63230185e+02
  2.59164737e+03]
 [1.00000000e+00 6.40500000e+00 4.10240250e+01 2.62758880e+02
  1.68297063e+03]
 [1.00000000e+00 5.97600000e+00 3.57125760e+01 2.13418354e+02
  1.27538808e+03]
 [1.00000000e+00 6.06500000e+00 3.67842250e+01 2.23096325e+02
  1.35307921e+03]
 [1.00000000e+00 6.14200000e+00 3.77241640e+01 2.31701815e+02
  1.42311255e+03]
 [1.00000000e+00 4.13800000e+00 1.71230440e+01 7.08551561e+01
  2.93198636e+02]
 [1.00000000e+00 6.40600000e+00 4.10368360e+01 2.62881971e+02
  1.68402191e+03]
 [1.00000000e+00 6.01500000e+00 3.61802250e+01 2.17624053e+02
  1.30900868e+03]
 [1.00000000e+00 5.96300000e+00 3.55573690e+01 2.12028591e+02
  1.26432649e+03]
 [1.00000000e+00 6.40200000e+00 4.09856040e+01 2.62389837e+02
  1.67981974e+03]
 [1.00000000e+00 5.41400000e+00 2.93113960e+01 1.58691898e+02
  8.59157935e+02]
 [1.00000000e+00 6.14400000e+00 3.77487360e+01 2.31928234e+02
  1.42496707e+03]
 [1.00000000e+00 6.45400000e+00 4.16541160e+01 2.68835665e+02
  1.73506538e+03]
 [1.00000000e+00 6.25000000e+00 3.90625000e+01 2.44140625e+02
  1.52587891e+03]
 [1.00000000e+00 5.81300000e+00 3.37909690e+01 1.96426903e+02
  1.14182959e+03]
 [1.00000000e+00 5.79000000e+00 3.35241000e+01 1.94104539e+02
  1.12386528e+03]
 [1.00000000e+00 5.78200000e+00 3.34315240e+01 1.93301072e+02
  1.11766680e+03]
 [1.00000000e+00 6.85200000e+00 4.69499040e+01 3.21700742e+02
  2.20429349e+03]
 [1.00000000e+00 6.67400000e+00 4.45422760e+01 2.97275150e+02
  1.98401435e+03]
 [1.00000000e+00 5.70900000e+00 3.25926810e+01 1.86071616e+02
  1.06228285e+03]
 [1.00000000e+00 5.96000000e+00 3.55216000e+01 2.11708736e+02
  1.26178407e+03]
 [1.00000000e+00 5.59700000e+00 3.13264090e+01 1.75333911e+02
  9.81343901e+02]
 [1.00000000e+00 6.28600000e+00 3.95137960e+01 2.48383722e+02
  1.56134007e+03]
 [1.00000000e+00 6.37700000e+00 4.06661290e+01 2.59327905e+02
  1.65373405e+03]
 [1.00000000e+00 7.48900000e+00 5.60851210e+01 4.20021471e+02
  3.14554080e+03]
 [1.00000000e+00 5.57000000e+00 3.10249000e+01 1.72808693e+02
  9.62544420e+02]
```

```
pred_uniRM_y_poly = polyReg.predict(X_test_poly)
```

Fit the two models and evaluate their MSE. Which model performs better and why?

```
from sklearn.metrics import mean_squared_error
print('Mean squared error of linear regression ', mean_squared_error(testY, pred_uniRM_y))
print('Mean squared error of 4th order polynominal regression ', mean_squared_error(testY, pred_uniRM_y_poly))
```

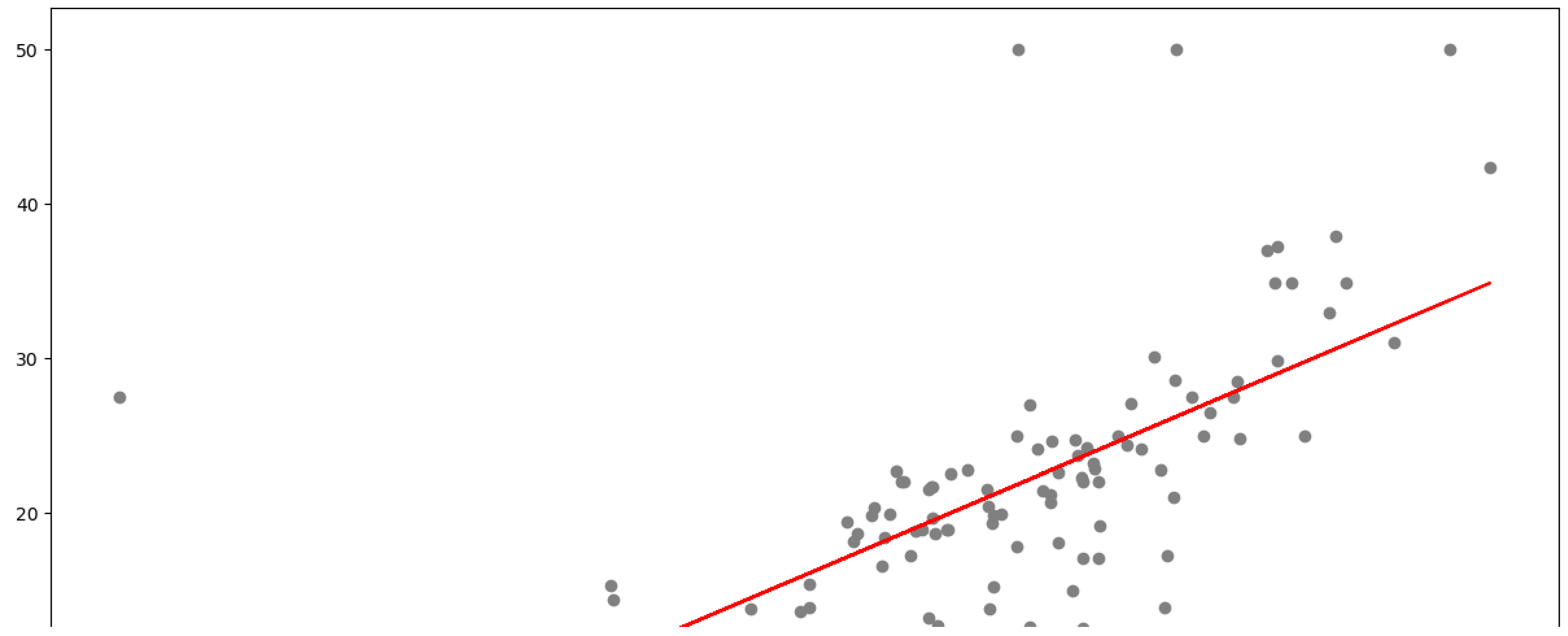
```
Mean squared error of linear regression  46.67135913580404
Mean squared error of 4th order polynominal regression  40.456829914611156
```

Double-click (or enter) to edit

Let’s do some dataviz to see what our model looks like

```
plt.figure(figsize=(15,9))

#plot the linear model
plt.scatter(testX, testY, color='grey')
plt.plot(testX, pred_uniRM_y, color='red', linewidth=1.5)

```

```
[<matplotlib.lines.Line2D at 0x1e138a5cd10>]
```

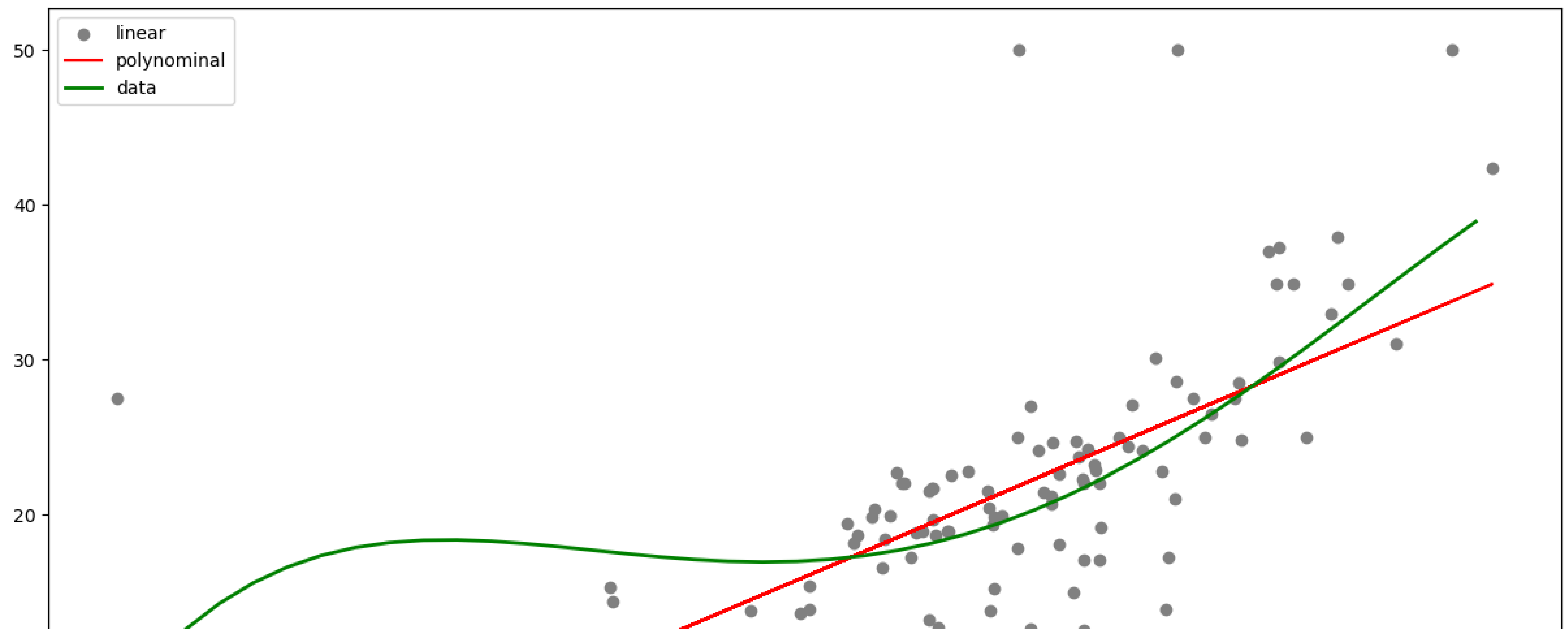


```
x_min = min(testX.iloc[:,:].values)[0]
x_max = max(testX.iloc[:,:].values)[0]
print(x_min, x_max)
X_grid=np.arange(x_min,x_max,0.1)
X_grid=X_grid.reshape((len(X_grid),1))
```

```
3.561 7.61
```

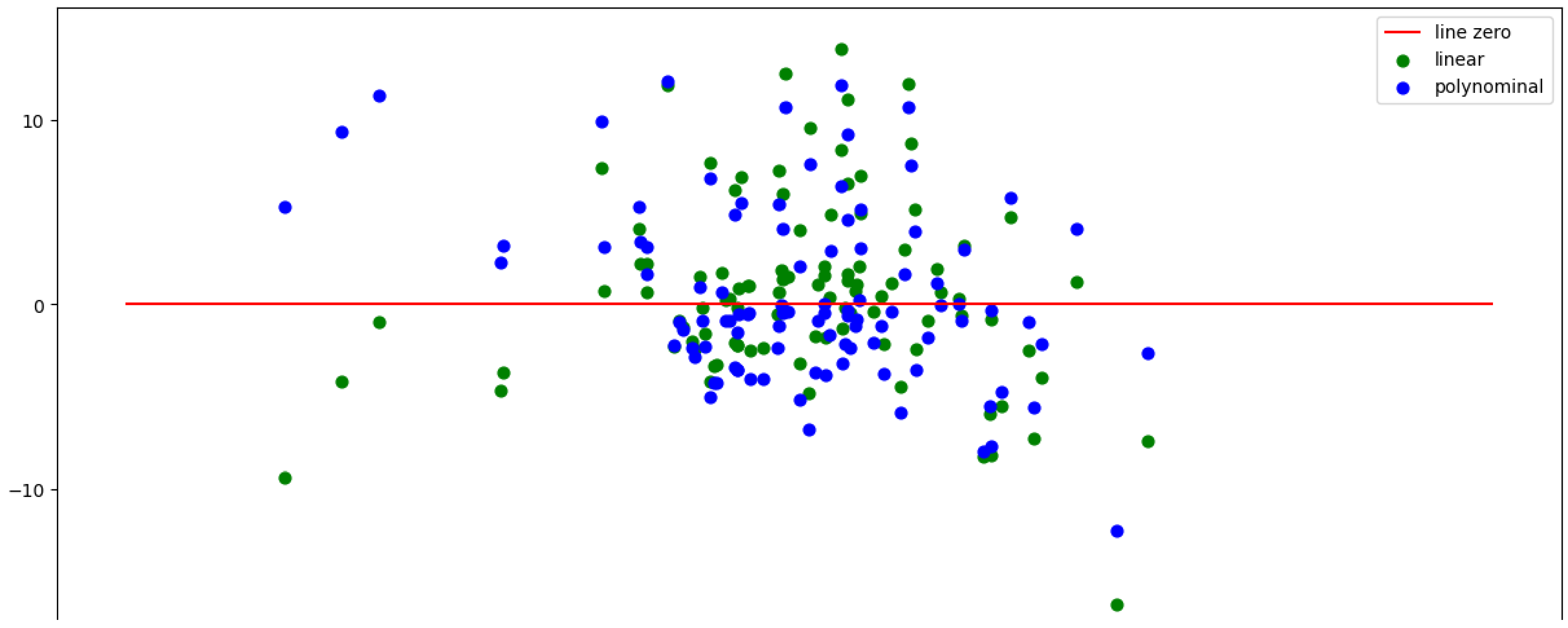
```
plt.figure(figsize=(15,9))

#plot the linear model
plt.scatter(testX, testY, color='grey')
plt.plot(testX, pred_uniRM_y, color='red', linewidth=1.5)

#plot the 4th order polynominal model
X_grid_poly= poly_feature.fit_transform(X_grid)
pred_uniRM_y_grid_poly = polyReg.predict(X_grid_poly)
plt.plot(X_grid, pred_uniRM_y_grid_poly, color='green', linewidth=2)
plt.legend(['linear','polynominal', 'data'])
```

```
<matplotlib.legend.Legend at 0x1e138acbe10>
```



```
plt.figure(figsize=(15,9))
plt.hlines(y=0, xmin=3.5, xmax=9, colors = 'red')
plt.scatter(testX, linReg.predict(testX) - testY, c='g', s=40)
plt.scatter(testX, pred_uniRM_y_poly - testY, c='b', s=40)
plt.legend(['line zero','linear', 'polynominal'])
```

```
<matplotlib.legend.Legend at 0x1e138a1cad0>
```



If the model is perfect, then for each testing data then there is 0 residual (or 0 error between predicted versus actual), which is represented by the horizontal line in the diagram. If the model under-estimates the actual value, it will be below this horizontal line, and if over-estimates, than above. This plot can quickly give you a sense of where the errors are occurring and whether there are outlier points that are causing large errors (we will examine this later in the course and ways to deal with this).

> **☞ With your dataframe slicing skill, can you count the number of residuals that are larger than zero and those that are smaller than zero?

```
positive_residuals = (pred_uniRM_y_poly - testY) > 0
negative_residuals = (pred_uniRM_y_poly - testY) < 0

num_positive_residuals = sum(positive_residuals)
num_negative_residuals = sum(negative_residuals)

print("Number of residuals larger than zero:", num_positive_residuals)
print("Number of residuals smaller than zero:", num_negative_residuals)
```

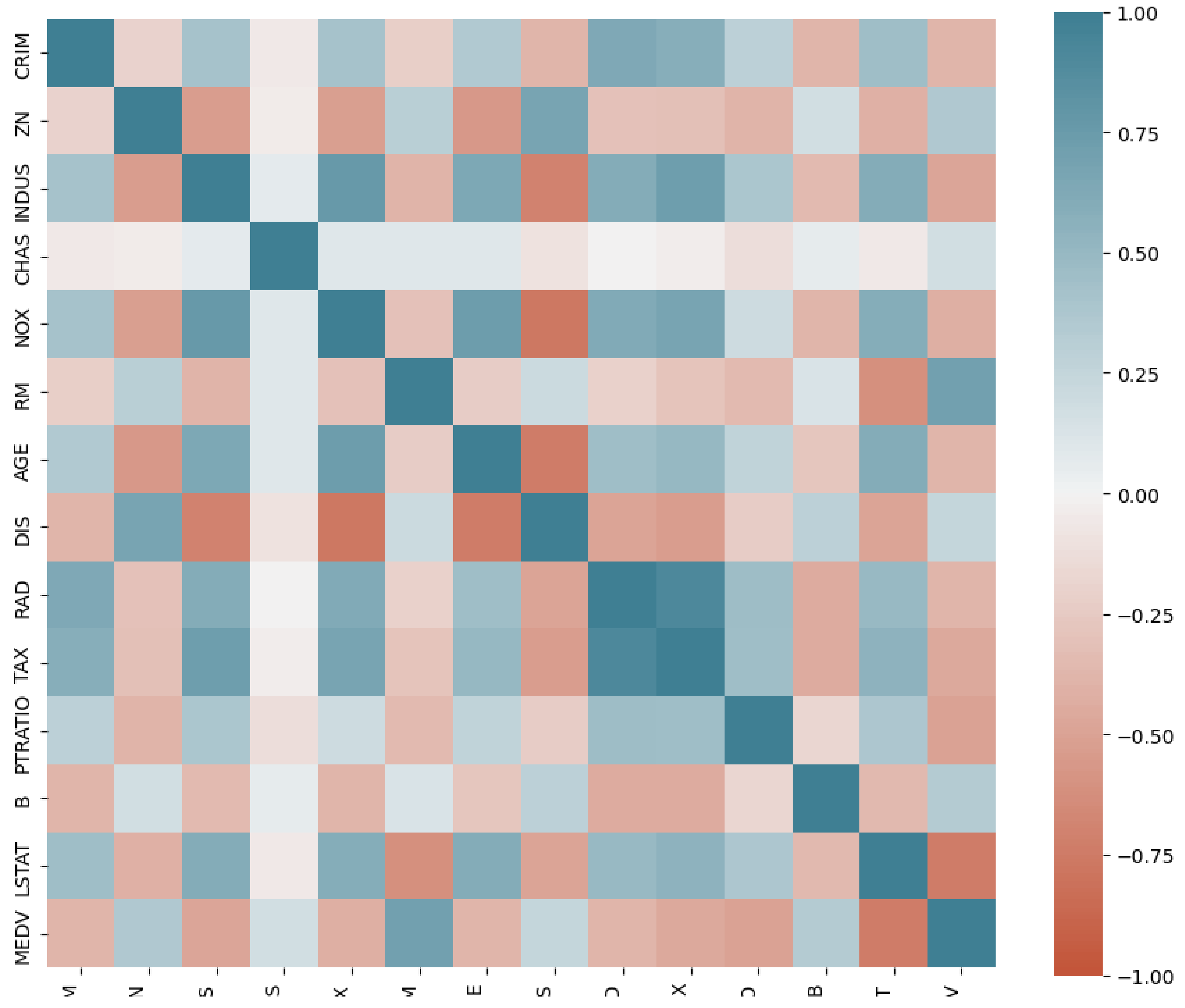
```
Number of residuals larger than zero: 39
Number of residuals smaller than zero: 63
```

Further exercies

We have fitted a linear regression model/hypothesis for the 'RM' variable. Use the seaborn package to visualise the correlation plot and identify the next potential predictor of the house price. You should identify 'LSTAT' as the next potential predictor. Why is it?

Repeat the above analysis using the 'LSTAT' variable, and comment on which predictor you think is performing better. LSTAT variable has a negative (inverse) linear relationship with 'MEDV', where when LSTAT increases MEDV decreases – do you think it matters?

```
import seaborn as sns

f, ax = plt.subplots(figsize=(11, 9))
corr = bostonHouseFrame.corr()
ax = sns.heatmap(
    corr,
    vmin=-1, vmax=1, center=0,
    cmap=sns.diverging_palette(20, 220, n=200),
    square=True
)
ax.set_xticklabels(
    ax.get_xticklabels(),
    rotation=90,
    horizontalalignment='right'
);
```



This Python code is using the seaborn and matplotlib libraries to create a heatmap of the correlation matrix for a dataframe `bostonHouseTrainFrame`.

The first line of code imports the seaborn library, which is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.

The second line of code creates a new figure and a single subplot. The `plt.subplots(figsize=(11, 9))` function is used to create the figure and subplot, and the `figsize` parameter is set to (11, 9), which controls the size of the figure.

The third line of code calculates the correlation matrix of the `bostonHouseTrainFrame` dataframe. The `corr()` function is used to compute pairwise correlation of columns, excluding NA/null values. The correlation coefficient ranges from -1 to 1. A value closer to 1 implies a strong positive correlation and a value closer to -1 implies a strong negative correlation.

The next block of code creates a heatmap of the correlation matrix using the seaborn function `sns.heatmap()`. The `vmin` and `vmax` parameters are set to -1 and 1, respectively, which set the color scale limits. The `center` parameter is set to 0, which centers the colormap at that value. The `cmap` parameter is set to a diverging palette, which means the colors diverge from the center value to the `vmin` and `vmax` values. The `square` parameter is set to `True`, which means each cell will be square-shaped.

The final block of code sets the x-axis tick labels. The `ax.get_xticklabels()` function is used to get the current x-axis tick labels, and the `rotation` parameter is set to 90, which means the labels will be rotated 90 degrees. The `horizontalalignment` parameter is set to 'right', which means the labels will be aligned to the right.

This heatmap is useful for quickly visualizing the relationships between each pair of variables. For example, if two variables have a strong positive correlation, their corresponding cell in the heatmap will be a bright color, and if they have a strong negative correlation, the cell will be a dark color.

Why 'LSTAT' as a Predictor?

Statistical Reasoning: Often in housing datasets, the socio-economic status of the neighborhood, represented by 'LSTAT', has a significant impact on house prices. Lower status of the population in an area could inversely affect the median value of houses, as these areas might be less desirable.

Negative Correlation with 'MEDV': A negative (inverse) relationship between 'LSTAT' and 'MEDV' suggests that as 'LSTAT' increases (indicating a higher percentage of lower-status population), 'MEDV' tends to decrease. This relationship can be a strong indicator that 'LSTAT' is a meaningful predictor for 'MEDV'.

We should identify 'LSTAT' as the next potential predictor because the coefficient is near -1

```
#print(bostonHouseFrame)
#house_uniRM_x = bostonHouseFrame.loc[:,'RM']
#house_uniRM_x= house_uniRM_x.values.reshape(-1,1)
house_uniRM_x = bostonHouseFrame[['LSTAT']]
house_y = bostonHouseFrame['MEDV']
```

```
# create testing and training data for RM variable
#from sklearn.model_selection import train_test_split
trainX, testX, trainY, testY = train_test_split(house_uniRM_x, house_y, test_size=0.2,shuffle=True)
print(trainX.shape)
print(testX.shape)
print(trainY.shape)
print(testY.shape)
```

```
(404, 1)
(102, 1)
(404,)
(102,)
```

```
from sklearn import linear_model
linReg = linear_model.LinearRegression()
linReg.fit(trainX, trainY)
```

```
▾ LinearRegression
LinearRegression()
```

```
print(linReg.intercept_)
print(linReg.coef_)
```

```
35.09864316713197
[-0.98351824]
```

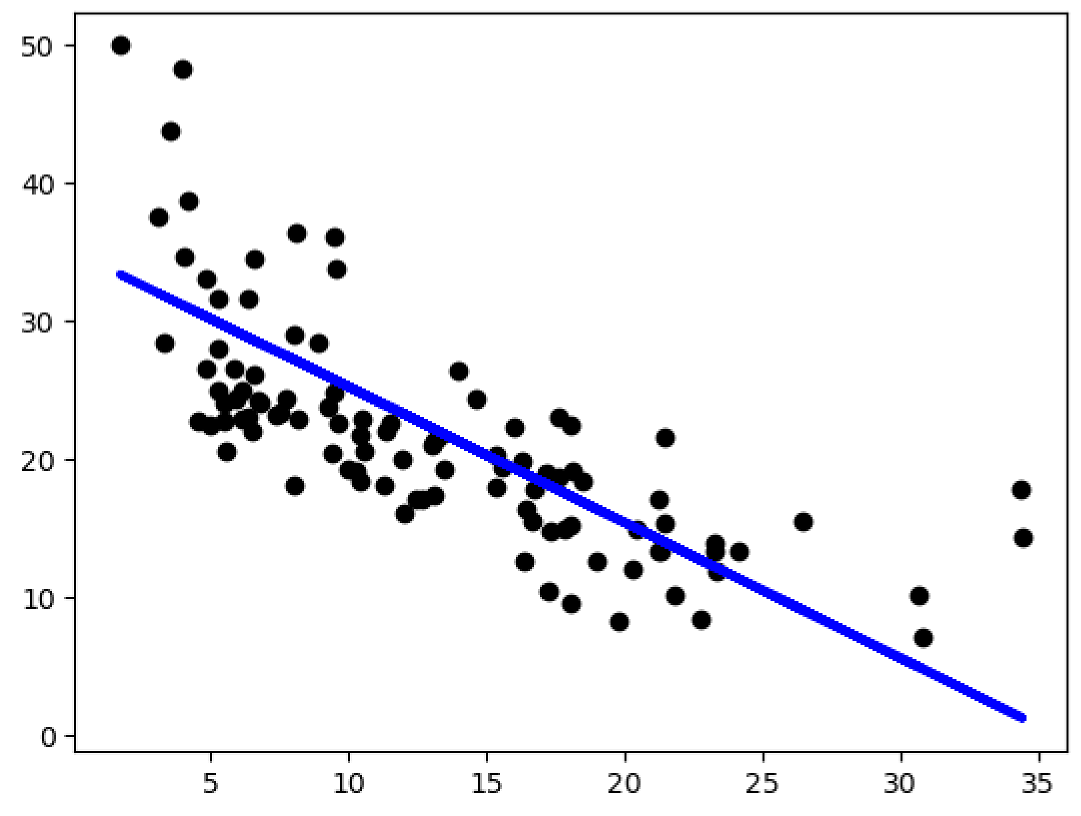
```
pred_uniRM_y = linReg.predict(testX)

```

```
from sklearn.metrics import mean_squared_error
print('Mean squared error ', mean_squared_error(testY, pred_uniRM_y))
```

```
Mean squared error  31.713395580333277
```

```
plt.scatter(testX, testY, color='black')
plt.plot(testX, pred_uniRM_y, color='blue', linewidth=3)
```

```
[<matplotlib.lines.Line2D at 0x1e137975a10>]
```



plt.scatter(testX, linReg.predict(testX) - testY, c='g', s=40) plt.hlines(y=0, xmin=0, xmax=35)

Multivariate Regression

So far, we have only used one X variable to predict MEDV. What if we used all the variables? To create such data, we first do the following:

```
house_multi_X = bostonHouseFrame.drop('MEDV', axis=1)
house_multi_Y = bostonHouseFrame[['MEDV']]
```

```
print(house_multi_X.shape)
print(house_multi_Y.shape)
```

```
(506, 13)
(506, 1)
```

```
train_multi_X, test_multi_X, train_multi_Y, test_multi_Y = train_test_split(house_multi_X, house_multi_Y, test_size=0.2,shuffle=True)
```

The Y variable is the same, but the X variable is interesting. We use the drop() method of data frames, which essentially drops a column from it – in our case, we drop the 'MEDV' column. X is now a data frame without the MEDV column, print it out the check.

Now repeat the same analysis, noting that train_test_split(), linReg.fit() and linReg.predict() can work with multivariate linear regression (i.e. where we have multiple X variables) as well. Make sure you print out the intercept and coefficients, print(linReg.intercept_) print(linReg.coef_), to see what are the fits and do the mean squared error.

```
from sklearn import linear_model
multi_linReg = linear_model.LinearRegression()
```

```
history = multi_linReg.fit(train_multi_X, train_multi_Y)
```

```
print(history)
```

```
LinearRegression()
```

```
print(multi_linReg.intercept_)
print(multi_linReg.coef_)
```

```
[35.87891027]
[[-1.05655638e-01  4.67639881e-02 -1.87121264e-02  1.80162716e+00
```

```
  -1.94097362e+01  4.00484229e+00 -1.28789329e-02 -1.49892551e+00
   2.58969412e-01 -1.20578490e-02 -8.91670461e-01  7.57585926e-03
  -4.16320997e-01]]
```

```

"""
Predicts the target variable using a multi-linear regression model.

Parameters:
test_multi_X (array-like): The input features for prediction.

Returns:
array-like: The predicted values of the target variable.
"""
pred_uniRM_y = multi_linReg.predict(test_multi_X)

```

```
from sklearn.metrics import mean_squared_error
print('Mean squared error ', mean_squared_error(test_multi_Y, pred_uniRM_y))
```

```
Mean squared error  37.99916867050114
```

Regularization - Lasso Polynomial Regression

> **☞ Goal: Do some regularization techniques on polynomial regression.

We will be using the Lasso polynomial regression model. Recall that Lasso regularisation (L1) has a regularisation weight that determines the weighting placed on regularisation. We use the following regularisation weights to evaluate which one is best for the regularisation weight in your constructed lasso polynomial regression.

- alpha = 0.01
- alpha = 0.05
- alpha = 0.1
- alpha = 0.25
- alpha = 0.5
- alpha = 0.75
- alpha = 1

Refer to the documentation about L1 regularisation to understand how to modify the polyminal regression model with the alpha parameter: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.Lasso.html

> **☞ Task: Build seven different Lasso polynomial regression models with the above alpha parameters and evaluate which one works best (based on MSE).**

> **☞ Question: Keep the order of polynomial the same (i.e., 4). What is the alpha parameter that leads to the best model?**

In order to do model selection with different regularisation alpha values, we first store the alpha values in a list:

```
alpha_RegPara = [0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1]
house_uniRM_x = bostonHouseFrame[['RM']]
house_y = bostonHouseFrame['MEDV']
```

```
#just remind that we already have X_train, Y_train
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(house_uniRM_x, house_y, test_size=0.2,shuffle=True)
```

First, we need to import Lasso from linear regesstion model

```
from sklearn.linear_model import Lasso
```

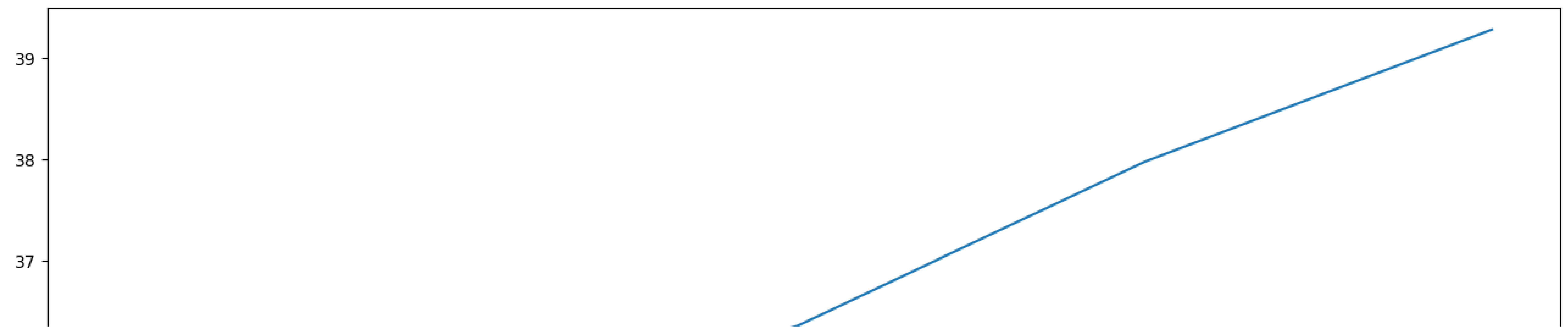
And prapare data for training and validation

```
# We need to have lists to store our results
final_results = []

# Create the train and the validation sets based on the existing training set
#for trainIndex, validIndex in kFold.split(X_train, Y_train):
# Use 20% of the existing train set to make the validation set
trainX = np.array(X_train.iloc[:])
trainY = np.array(Y_train.iloc[:])
validX = np.array(X_train.iloc[:])
validY = np.array(Y_train.iloc[:])

print(trainX.shape)
print(validX.shape)
#print(trainX)
#print(validX)

```

```
(404, 1)
(404, 1)
```

```python
from sklearn.preprocessing import StandardScaler

# We have the training and validation data now
# We gonna train each model with each alphas by using these training data
# And then we are going to evaluate MSE of the models by applying them on the validation data
# To store the results of each alphas
lResults = []
lModels = []
scaler = StandardScaler()

for regPara in alpha_RegPara:
    ### TRAINING THE LASSO MODEL
    # Create the polynomial regression object
    polyLassoReg = Lasso(alpha = regPara)

    # Create, then fit and transform at the same time trainX by using the poly_feat object
    trainX_scaled = scaler.fit_transform(trainX)
    polyFitTrainX = poly_feature.fit_transform(trainX_scaled)

    # Fit the model
    polyLassoReg.fit(polyFitTrainX, trainY)

    ### VALIDATE THE LASSO MODEL WITH THE VALIDATION DATA
    # Create, then fit and transform at the same time validationX by using the poly_feat object
    validX_scaled = scaler.transform(validX)
    polyFitValidX = poly_feature.fit_transform(validX_scaled)

    # Predict Y by using the validation set
    predY = polyLassoReg.predict(polyFitValidX)

    # Calculate the MSE
    mse = mean_squared_error(predY, validY)

    # Store the MSE result of the current model
    lResults.append(mse)

print(alpha_RegPara)
print("-----------------------------------------------------")
print("")
final_results.append(lResults)
plt.figure(figsize=(16,5))
plt.plot(alpha_RegPara, lResults)
```

```
[0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1]
-----------------------------------------------------

[<matplotlib.lines.Line2D at 0x1e13a321890>]
```



K-Fold Cross Validation

We will now use k-fold cross validation to evaluate parameters and to perform testing performance.

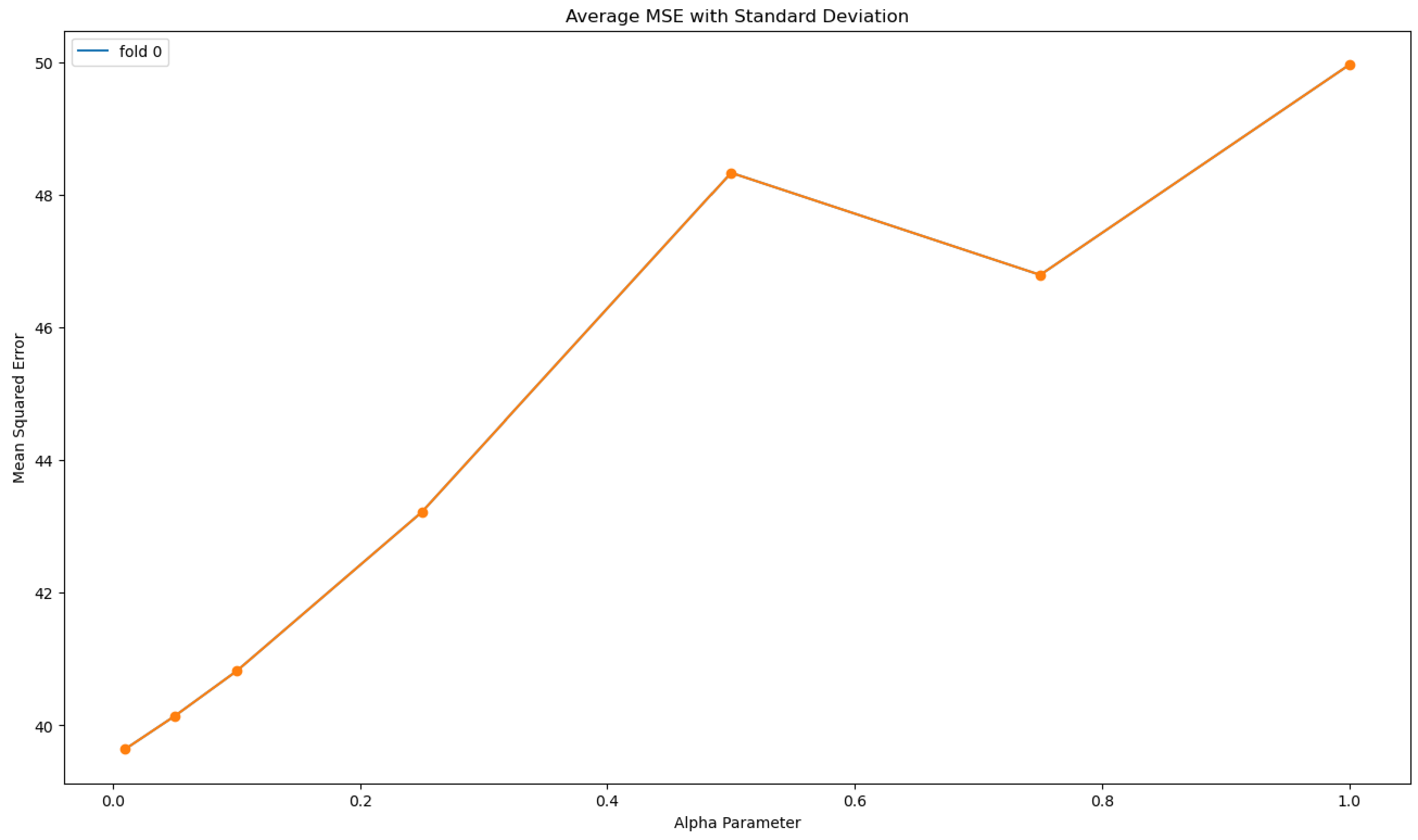
Consider the following code to setup 5-fold cross validation:

```python
from sklearn import model_selection

kFold = model_selection.KFold(n_splits=5, shuffle=True)
```

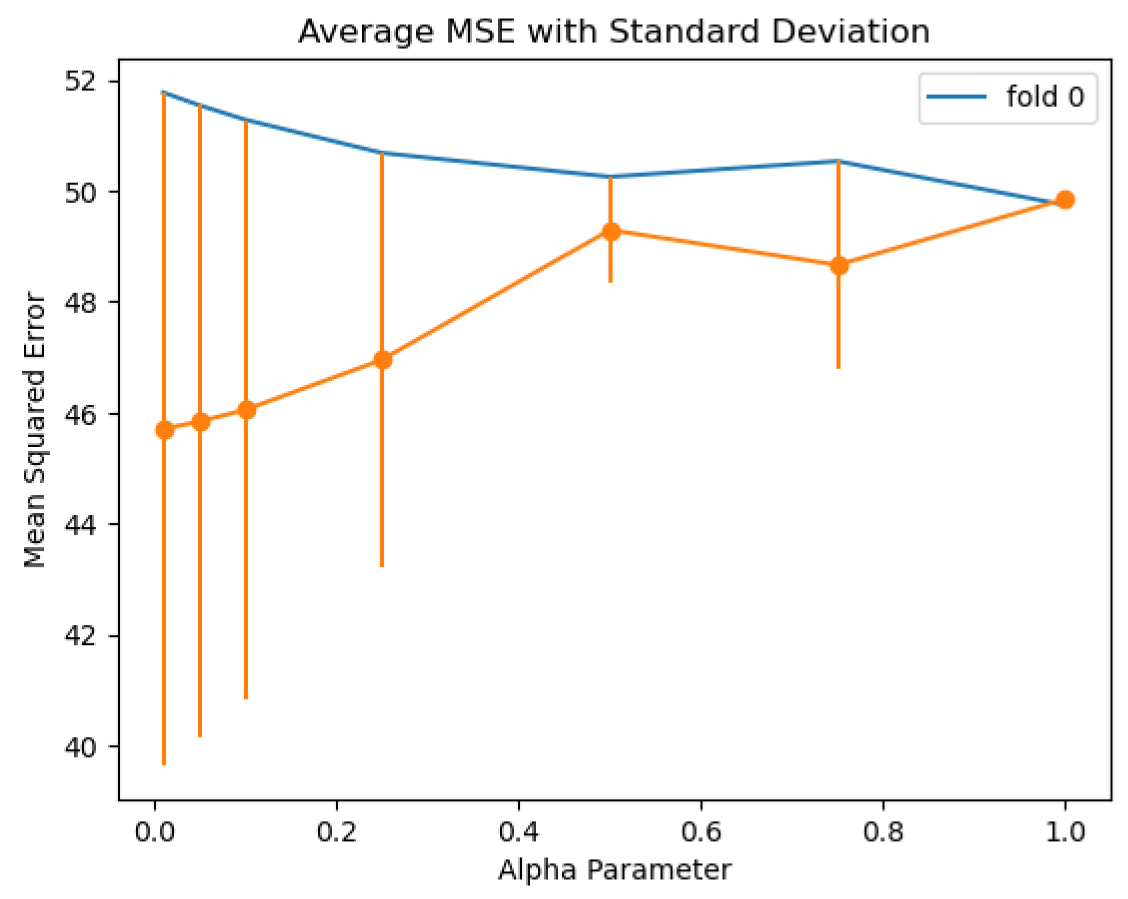
The shuffle parameter of KFold() randomly shuffles the data before performing the splits. You may want to do some extra reading to understand how to use KFold().

```python
from sklearn.preprocessing import StandardScaler

# We need to have lists to store our results
final_results = []
i = 0
plt.figure(figsize=(16,9))
scaler = StandardScaler()

# Create the train and the validation sets based on the existing training set
for trainIndex, validIndex in kFold.split(X_train, Y_train):
    # Use 20% of the existing train set to make the validation set
    trainX = np.array(X_train.iloc[trainIndex])
    trainY = np.array(Y_train.iloc[trainIndex])
    validX = np.array(X_train.iloc[validIndex])
    validY = np.array(Y_train.iloc[validIndex])

    # We have the training and validation data now
    # We gonna train each model with each alphas by using these training data
    # And then we are going to evaluate MSE of the models by applying them on the validation data

    # To store the results of each alphas
    lResults = []
    lModels = []

    for regPara in alpha_RegPara:
        ### TRAINING THE LASSO MODEL
        # Create the polynomial regression object
        polyLassoReg = Lasso(alpha = regPara)

        # Create, then fit and transform at the same time trainX by using the poly_feat object
        trainX_scaled = scaler.fit_transform(trainX)
        polyFitTrainX = poly_feature.fit_transform(trainX_scaled)

        # Fit the model
        polyLassoReg.fit(polyFitTrainX, trainY)

        ### VALIDATE THE LASSO MODEL WITH THE VALIDATION DATA
        # Create, then fit and transform at the same time validationX by using the poly_feat object
        validX_scaled = scaler.transform(validX)
        polyFitValidX = poly_feature.fit_transform(validX_scaled)

        # Predict Y by using the validation set
        predY = polyLassoReg.predict(polyFitValidX)

        # Calculate the MSE
        mse = mean_squared_error(predY, validY)

        # Store the MSE result of the current model
        lResults.append(mse)

    i = i + 1

    print(alpha_RegPara)
    print("Fold",i,lResults)
    print("---------------------------------------------------")
    print("")
    final_results.append(lResults)
    plt.plot(alpha_RegPara, lResults)
    plt.legend(['fold ' + str(i) for i in range(5)])

    # Calculate the average and standard deviation of the results
    avg_results = np.mean(final_results, axis=0)
    std_results = np.std(final_results, axis=0)

    # Print the average and standard deviation
    print("Average MSE:", avg_results)
    print("Standard Deviation:", std_results)

    # Plot the average results with error bars representing the standard deviation
    plt.errorbar(alpha_RegPara, avg_results, yerr=std_results, fmt='-o')

    # Set the labels and title of the plot
    plt.xlabel('Alpha Parameter')
    plt.ylabel('Mean Squared Error')
    plt.title('Average MSE with Standard Deviation')

    # Show the plot
    plt.show()
```

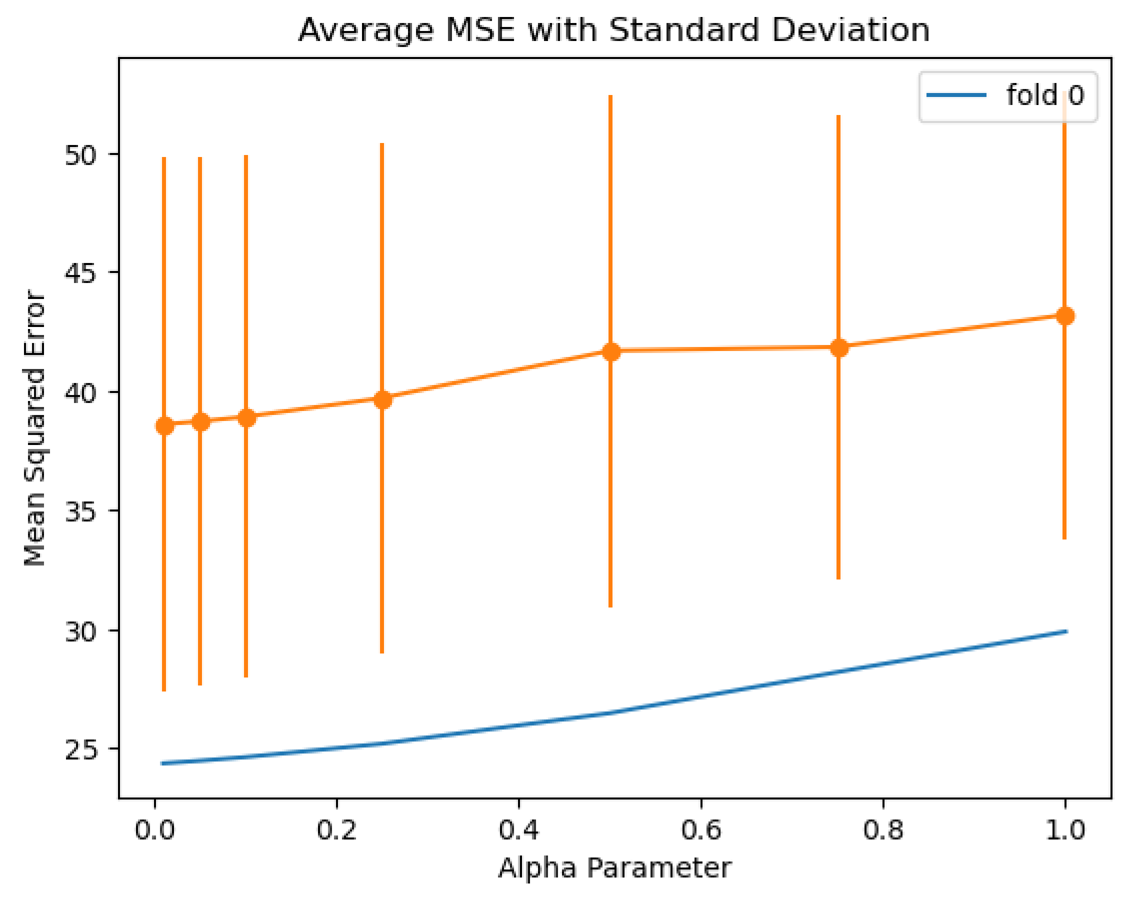
```
[0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1]
Fold 1 [39.64377497674707, 40.14186341204853, 40.819489366118326, 43.22064237841496, 48.3349353831262, 46.79413000091956, 49.964037102650444]
----------------------------------------------------

Average MSE: [39.64377498 40.14186341 40.81948937 43.22064238 48.33493538 46.79413
 49.9640371 ]
Standard Deviation: [0. 0. 0. 0. 0. 0. 0.]
```



```
[0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1]
Fold 2 [51.771331940458545, 51.539239350023934, 51.28072729163573, 50.68165412737446, 50.25074601698898, 50.53214822582273, 49.74000481404778]
----------------------------------------------------

Average MSE: [45.70755346 45.84055138 46.05010833 46.95114825 49.2928407  48.66313911
 49.85202096]
Standard Deviation: [6.06377848 5.69868797 5.23061896 3.73050587 0.95790532 1.86900911
 0.11201614]
```

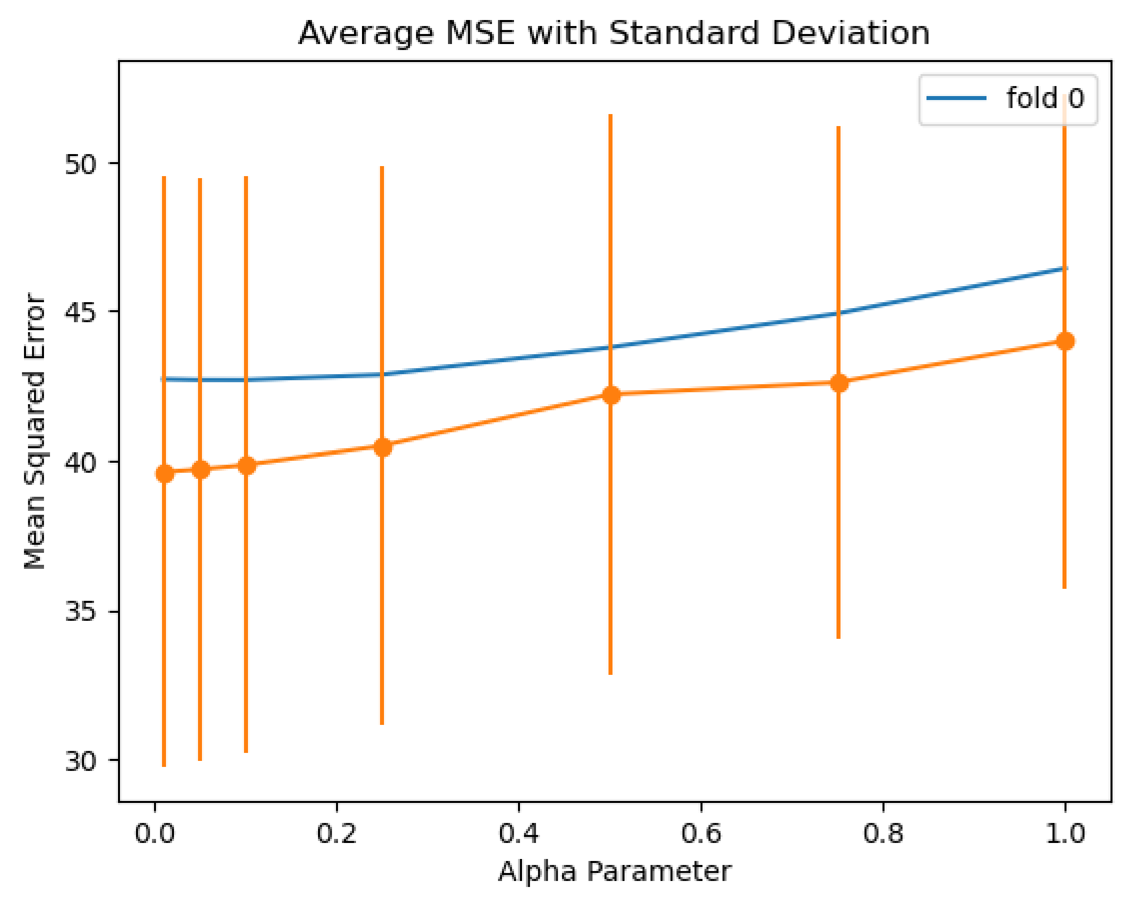


```
[0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1]
Fold 3 [24.362582944952084, 24.47201561168461, 24.62438957910274, 25.186313180855155, 26.47374926713303, 28.200502299253234, 29.899512171127707]
----------------------------------------------------

Average MSE: [38.59256329 38.71770612 38.90820208 39.69620323 41.68647689 41.84226018
 43.2011847 ]
Standard Deviation: [11.21423693 11.09593974 10.96600351 10.70262759 10.78541898  9.76614436
  9.40614751]
```

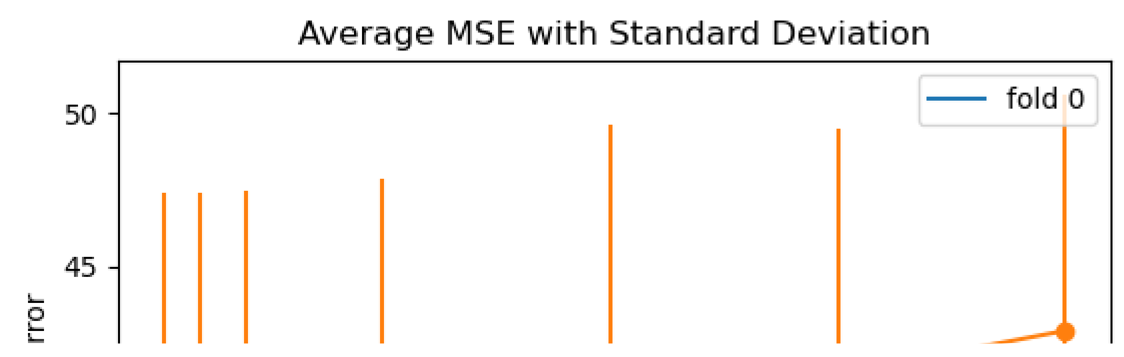


```
[0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1]
Fold 4 [42.71796860676363, 42.69525194760479, 42.69297299169345, 42.86977801006112, 43.78207346872678, 44.918385502553704, 46.43304729295807]
----------------------------------------------------

Average MSE: [39.62391462 39.71209258 39.85439481 40.48959692 42.21037603 42.61129151
```

```
 44.00915035]
Standard Deviation: [9.87473489 9.7624957  9.63720741 9.37006396 9.38442102 8.56197462
 8.26529696]
```



```
[0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1]
Fold 5 [31.553090237343916, 31.696597674609098, 31.899668477107333, 32.66697851682794, 34.47310637121093, 36.93793762478989, 38.419069463884476]
---------------------------------------------------

Average MSE: [38.00974974 38.1089936  38.26344954 38.92507324 40.6629221  41.47662073
 42.89113417]
Standard Deviation: [9.40374523 9.30186881 9.18831    8.94591618 8.94607964 7.98722972
 7.72346734]
```



```
print(['fold ' + str(i) for i in range(5)])
```

```
['fold 0', 'fold 1', 'fold 2', 'fold 3', 'fold 4']
```

Modify the code above to output an average (and std. dev.) of the results and plot it. Use this result to help select the regularisation parameter setting.

> ☞ **What is the alpha value that we should use for our Lasso polynomial model and why?**

> ☞**Your next task is to adjust the order of polynomial to 2 and 3 and evaluate the results. Do these models perform better than the order 4 polynomial model?**

To determine the optimal alpha value for a Lasso polynomial model, you need to perform cross-validation. The process involves:

1. Using `PolynomialFeatures` to transform your data into the desired polynomial degree.
2. Applying `LassoCV` from `sklearn` to perform Lasso regression across a range of alpha values.
3. The `LassoCV` will automatically choose the alpha that results in the best cross-validation score.

The selected alpha is optimal because it achieves the best trade-off between fitting the training data and maintaining the model's ability to generalize to new, unseen data. This helps to prevent overfitting, where the model learns the noise in the training data rather than the underlying trend.

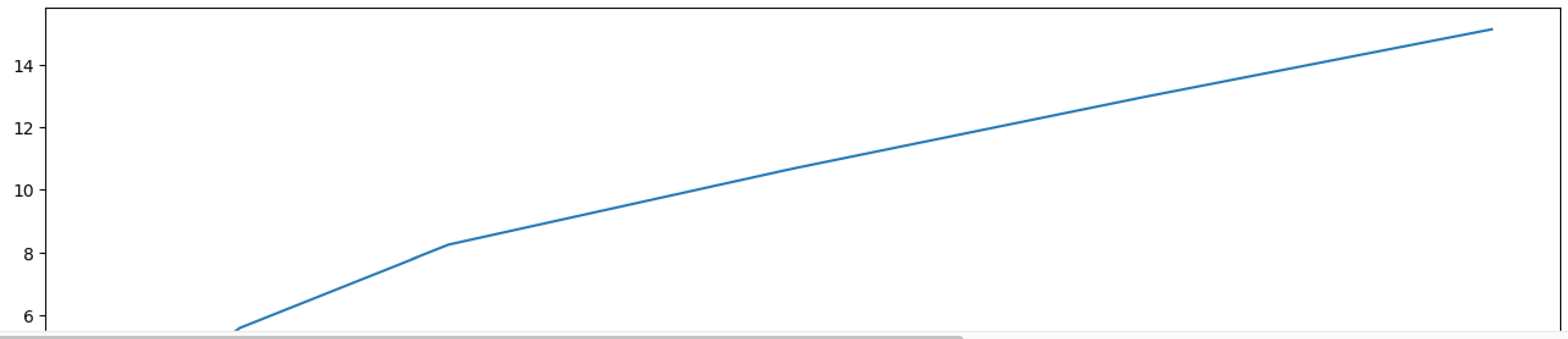
```python
from sklearn.datasets import make_regression
from sklearn.linear_model import LassoCV
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures
from sklearn.metrics import mean_squared_error
from sklearn.pipeline import make_pipeline

# Generate a synthetic regression dataset
X, y = make_regression(n_samples=506, n_features=13, noise=0.1, random_state=42)

# Split data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)

# Create a pipeline that creates polynomial features and then applies LassoCV
# Let's start with a polynomial degree of 2
degree = 2
lasso_poly2_model = make_pipeline(PolynomialFeatures(degree), LassoCV(cv=5))

# Fit the model to the training data
lasso_poly2_model.fit(X_train, y_train)

# Extract the optimal alpha value for degree 2
optimal_alpha_poly2 = lasso_poly2_model.named_steps['lassocv'].alpha_

# Evaluate the model performance for degree 2
mse_poly2 = mean_squared_error(y_test, lasso_poly2_model.predict(X_test))

# Now repeat for a polynomial degree of 3
degree = 3
lasso_poly3_model = make_pipeline(PolynomialFeatures(degree), LassoCV(cv=5))
lasso_poly3_model.fit(X_train, y_train)

# Extract the optimal alpha value for degree 3
optimal_alpha_poly3 = lasso_poly3_model.named_steps['lassocv'].alpha_

# Evaluate the model performance for degree 3
mse_poly3 = mean_squared_error(y_test, lasso_poly3_model.predict(X_test))

print(f"Optimal alpha for degree 2 polynomial model: {optimal_alpha_poly2}, MSE: {mse_poly2}")
print(f"Optimal alpha for degree 3 polynomial model: {optimal_alpha_poly3}, MSE: {mse_poly3}")

```

```
Optimal alpha for degree 2 polynomial model: 0.08980600464878032, MSE: 0.0945788585545689
Optimal alpha for degree 3 polynomial model: 0.319856538640631, MSE: 3.406297699976902
```

> ☞ **Task: Work with multivariate regresstion!!!**

```python
alpha_RegPara = [0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1]

house_multi_X = bostonHouseFrame.drop('MEDV', axis=1)
house_multi_Y = bostonHouseFrame[['MEDV']]

train_multi_X, test_multi_X, train_multi_Y, test_multi_Y = train_test_split(house_multi_X, house_multi_Y, test_size=0.2,shuffle=True)

from sklearn.linear_model import Lasso

# We need to have lists to store our results
final_results = []

# Create the train and the validation sets based on the existing training set
#for trainIndex, validIndex in kFold.split(X_train, Y_train):
# Use 20% of the existing train set to make the validation set
trainX = np.array(train_multi_X.iloc[:])
trainY = np.array(train_multi_Y.iloc[:])
validX = np.array(train_multi_X.iloc[:])
validY = np.array(train_multi_Y.iloc[:])

print(trainX.shape)
print(validX.shape)
#print(trainX)
#print(validX)

from sklearn.preprocessing import StandardScaler

# We have the training and validation data now
# We gonna train each model with each alphas by using these training data
# And then we are going to evaluate MSE of the models by applying them on the validation data
# To store the results of each alphas
lResults = []
lModels = []
scaler = StandardScaler()

for regPara in alpha_RegPara:
    ### TRAINING THE LASSO MODEL
    # Create the polynomial regression object
    polyLassoReg = Lasso(alpha = regPara, max_iter=10000)


    # Create, then fit and transform at the same time trainX by using the poly_feat object
    trainX_scaled = scaler.fit_transform(trainX)
    polyFitTrainX = poly_feature.fit_transform(trainX_scaled)

    # Fit the model
    polyLassoReg.fit(polyFitTrainX, trainY)

    ### VALIDATE THE LASSO MODEL WITH THE VALIDATION DATA
    # Create, then fit and transform at the same time validationX by using the poly_feat object
    validX_scaled = scaler.transform(validX)
    polyFitValidX = poly_feature.fit_transform(validX_scaled)

    # Predict Y by using the validation set
    predY = polyLassoReg.predict(polyFitValidX)

    # Calculate the MSE
    mse = mean_squared_error(predY, validY)

    # Store the MSE result of the current model
    lResults.append(mse)

print(alpha_RegPara)
print("----------------------------------------------------")
print("")
final_results.append(lResults)
plt.figure(figsize=(16,5))
plt.plot(alpha_RegPara, lResults)


```

```
(404, 13)
(404, 13)
c:\Users\Surface1\Anaconda3\Lib\site-packages\sklearn\linear_model\_coordinate_descent.py:628: ConvergenceWarning: Objective did not converge. You might want to increase t
  model = cd_fast.enet_coordinate_descent(
c:\Users\Surface1\Anaconda3\Lib\site-packages\sklearn\linear_model\_coordinate_descent.py:628: ConvergenceWarning: Objective did not converge. You might want to increase t
  model = cd_fast.enet_coordinate_descent(
[0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1]
----------------------------------------------------

[<matplotlib.lines.Line2D at 0x1e13bca66d0>]
```



Exercise: Work on the Bike Share Data

> ☞ **Task: Do the linear regression on the Bike Share Data.**
> Now you seen how to do this task for the house price dataset. Repeat the same process for the Daily Bike Share rental data.

1. Load the Daily Bike Share rental data.

2. Preprocess the data if necessary (handle missing values, encode categorical variables, etc.).
3. Split the data into training and testing sets.
4. Standardize or normalize the features if necessary.
5. Define a range of alpha parameters for the Lasso regression model.
6. For each alpha parameter, fit the Lasso model on the training data and evaluate its performance on the testing data.
7. Plot the performance of the model (e.g., MSE) against the alpha parameters.
8. Choose the alpha parameter that gives the best performance.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler
# Read the Bike Share data
bike_data = pd.read_csv('bikeShareDay-1.csv')

# Split the data into features (X) and target (Y)
bike_X = bike_data.drop(['cnt', 'dteday'], axis=1)
bike_Y = bike_data['cnt']

print(bike_multi_X.shape)
print(bike_multi_Y.shape)

# Split the data into training and testing sets
X_train, X_test, Y_train, Y_test = train_test_split(bike_X, bike_Y, test_size=0.2, shuffle=True, random_state=42)

scaler = StandardScaler()

# Fit the scaler on the training data and transform both the training and testing data
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)


# Create a linear regression model
model = LinearRegression()

# Fit the model to the training data
model.fit(X_train, Y_train)

# Predict the target variable for the test data
Y_pred = model.predict(X_test)

# Calculate the mean squared error
mse = mean_squared_error(Y_test, Y_pred)

# Print the mean squared error
print("Mean Squared Error:", mse)

```

```
(731, 14)
(731, 1)
Mean Squared Error: 3.099113757575374e-24
```

```
import numpy as np
from sklearn.linear_model import Lasso
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

alpha_RegPara = [0.01, 0.05, 0.1, 0.25, 0.5, 0.75, 1]

# We need to have lists to store our results
final_results = []

# Create the train and the validation sets based on the existing training set
#for trainIndex, validIndex in kFold.split(X_train, Y_train):
# Use 20% of the existing train set to make the validation set
trainX = np.array(X_train[:])
trainY = np.array(Y_train[:])
validX = np.array(X_train[:])
validY = np.array(Y_train[:])

print(trainX.shape)
print(validX.shape)
#print(trainX)
#print(validX)

```

```
(584, 14)
(584, 14)
```

```
from sklearn.preprocessing import StandardScaler

# We have the training and validation data now
# We gonna train each model with each alphas by using these training data
# And then we are going to evaluate MSE of the models by applying them on the validation data
# To store the results of each alphas
lResults = []
lModels = []
scaler = StandardScaler()

for regPara in alpha_RegPara:
```