Name: Karim Mahmoud Abdelhamid

ID: 40-4739

First part: Using Sci-kit learn library

Second part: Implementing it from scratch without using any libriaries.

First part: Using Sci-kit learn library

Linear Rgression

```
In [1]:
1 import numpy as np
2 import pandas as pd
```

We are importing numpy and pandas libraries to use them in order to handle the datasets.

```
In [2]:
1 dataset = pd.read_csv("house_prices_data_training_data.csv")
```

We are reading the dataset from a file and saving it as a data frame in a variable called “Dataset”

```
In [3]:
1 dataset.describe()
```

Out[3]:

	id	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	condition	
count	1.799900e+04	1.799900e+04	17999.000000	17999.000000	17999.000000	1.799900e+04	17999.000000	17999.000000	17999.000000	17999.000000	17
mean	4.567994e+09	5.330743e+05	3.362965	2.061601	2051.822323	1.575873e+04	1.433163	0.007834	0.242458	3.448525	
std	2.862810e+09	3.644122e+05	0.934032	0.758632	902.744154	4.245514e+04	0.509663	0.088164	0.778857	0.667039	
min	1.000102e+06	7.500000e+04	0.000000	0.000000	370.000000	5.200000e+02	1.000000	0.000000	0.000000	1.000000	
25%	2.123049e+09	3.150000e+05	3.000000	1.500000	1410.000000	5.450500e+03	1.000000	0.000000	0.000000	3.000000	
50%	3.905121e+09	4.470000e+05	3.000000	2.000000	1900.000000	7.904000e+03	1.000000	0.000000	0.000000	3.000000	
75%	7.298010e+09	6.354750e+05	4.000000	2.500000	2500.000000	1.109600e+04	2.000000	0.000000	0.000000	4.000000	
max	9.900000e+09	7.700000e+06	33.000000	8.000000	13540.000000	1.651359e+06	3.500000	1.000000	4.000000	5.000000	

We are exploring the dataset (Exploratory data analysis) to get some information about the data such as “mean”, “standard deviation”, etc.

```
In [5]:
1 dataset.dropna(inplace=True)
2 X = dataset.iloc[:, 3:].values
3 y = dataset.iloc[:, 2].values
```

We are splitting the dataset into features (X) and output (Y), and reading their specific columns/rows from the data file.

.dropna function: Drops any row that has empty data.

```
In [6]:
1 from sklearn.model_selection import train_test_split
2 X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_state = 0)
```

I'm splitting my dataset into 2 parts. The first part (X_train, y_train) has 70% of the data and is used to produce a learning algorithm. The second part is t(X_test, y_test) has 30% of the data so we can use the trained model to predict y_test.

```
In [7]:
1 from sklearn.linear_model import LinearRegression
2 regressor = LinearRegression()
3 regressor.fit(X_train, y_train)

Out[7]: LinearRegression()
```

We are initializing the linear regression and we make it train on the two parameters (X_train, y_train)

```
In [8]:
1 y_pred = regressor.predict(X_test)
2
```

Here we are using the X_test to predict the output using the linear regression model that We've previously trained, and saving it as y_pred.

```
In [9]:
1 print(y_pred)

[334437.9487808  711170.71670073 608717.28330945 ... 405833.08724417
 305690.82933908 381003.75410789]
```

Here we are printing the predicted output y.

```
In [13]:
1 from sklearn.metrics import r2_score
2 print(r2_score(y_test, y_pred)*100,"%")

69.40685075841238 %
```

We are comparing the predicted y to the original y_test, and checking the model accuracy in percentage.

```
In [17]:
1 from sklearn.linear_model import Ridge
2 RegularizedRegressor = Ridge(alpha=500.0, fit_intercept=True, normalize=False, copy_X=True, max_iter=None, tol=0.001, so
3 RegularizedRegressor.fit(X_train, y_train)

Out[17]: Ridge(alpha=500.0)
```

Instead of the normal linear regression model, we are using the regularized linear repressor model with Alpha=500.0, and without any normalization.

In [20]:

```
y_pred = RegularizedRegressor.predict(X_test)
print(r2_score(y_test, y_pred)*100,"%")
```

```
65.9355673207749 %
```

We are comparing the predicted y to the original y_test, and checking the model accuracy in percentage in the regularized regression.

In [34]:

```
def calc_acc(acc_score):
    print('accuracies ',(acc_score))
    print("Maximum accuracy", acc_score[np.argmax(np.array(acc_score))]*100,"%"," fold #",np.argmax(np.array(acc_score)))
```

When we will use the K fold cross validation method, we will use this function calc_acc to print all the K folds' accuracy and the best accuracy.

In [35]:

```
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score

kf = KFold(n_splits=20, random_state=None)
model = LinearRegression()

acc_score = []

for train_index , test_index in kf.split(X):
    X_train , X_test = X[train_index,:],X[test_index,:]
    y_train , y_test = y[train_index] , y[test_index]

    model.fit(X_train,y_train)
    y_pred = model.predict(X_test)
    acc = r2_score(y_test, y_pred)
    acc_score.append(acc)


calc_acc(acc_score)
```

```
accuracies  [0.7124656930001974, 0.6690636181322779, 0.748084462236993, 0.7114516661296724, 0.6552661064123207, 0.6778421399
690185, 0.7076547326200006, 0.7357075967454112, 0.7005553087867069, 0.6016541959321406, 0.6657881029890469, 0.71047029146252
96, 0.7225136011388766, 0.6952228194074015, 0.7046145307502706, 0.6961438958321602, 0.7352233610926144, 0.7012585612989968,
0.716423889829682, 0.6957265728792347]
Maximum accuracy 74.80844622369929 %  fold # 2
```

We are splitting the data into 20 fold without any data shuffling. We are splitting it 20 times into 70% train sets and 30% test sets, then I will use each fold to train the data and save the accuracy in an array and find the best accuracy. In this case fold 2 has the best accuracy of 74.808446% using the linear regression model.

In [36]:

```
from sklearn.model_selection import KFold
from sklearn.metrics import accuracy_score

kf = KFold(n_splits=20, random_state=None)
model = Ridge(alpha=500.0, fit_intercept=True, normalize=False, copy_X=True, max_iter=None, tol=0.001, solver='auto', ra
model.fit(X_train, y_train)

acc_score = []

for train_index , test_index in kf.split(X):
    X_train , X_test = X[train_index,:],X[test_index,:]
    y_train , y_test = y[train_index] , y[test_index]

    model.fit(X_train,y_train)
    y_pred = model.predict(X_test)
    acc = r2_score(y_test, y_pred)
    acc_score.append(acc)

calc_acc(acc_score)
```

```
accuracies  [0.675913440075104, 0.6286829338838245, 0.6924410327399069, 0.6860623488267377, 0.6224607302763858, 0.6351013212
574989, 0.6924243901243878, 0.69341562251065, 0.6734924384542074, 0.5912126311208884, 0.6543032111849414, 0.662929767388620
3, 0.6838190298209212, 0.6742943179045404, 0.6920710252731139, 0.664770334369139, 0.7088933846550604, 0.6669947229071853, 0.
6993247467801744, 0.6776199558396137]
Maximum accuracy 70.88933846550603 %  fold # 16
```

We are splitting the data into 20 fold without any data shuffling. We are splitting it 20 times into 70% train sets and 30% test sets, then I will use each fold to train the data and save the accuracy in an array and find the best accuracy. In this case fold 16 has the best accuracy of 70.88934% using the regularized linear regression model. Alpha=500, and without any normalization.

In [38]:

```
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits=20)
skf.get_n_splits(X, y)
acc_score = []
model = LinearRegression()

print(skf)
StratifiedKFold(n_splits=5, random_state=None, shuffle=False)
for train_index, test_index in skf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    model.fit(X_train,y_train)
    y_pred = model.predict(X_test)
    acc = r2_score(y_test, y_pred)
    acc_score.append(acc)


calc_acc(acc_score)
```

```
StratifiedKFold(n_splits=20, random_state=None, shuffle=False)
accuracies  [0.7359213228364587, 0.6628566768781337, 0.7140534242381169, 0.6734612476949635, 0.6464476168478774, 0.669722167
0867031, 0.7156729993068361, 0.6863749507874504, 0.7122888154684228, 0.7008990925037357, 0.7360708951513714, 0.7326997150145
846, 0.7129854871459747, 0.7037736006500753, 0.7772738036005155, 0.6795367312500542, 0.6839675761098226, 0.7002678950435915,
0.6815019694388735, 0.6595979287571914]
Maximum accuracy 77.72738036005154 %  fold # 14
```

```
X:\Dev\Apps\Anaconda3\lib\site-packages\sklearn\model_selection\_split.py:670: UserWarning: The least populated class in y h
as only 1 members, which is less than n_splits=20.
  warnings.warn(("The least populated class in y has only %d"
```

We are using stratified K fold with linear regression with 20 folds. The best accuracy in fold number 14, and best accuracy of 77.7273%

In [39]:

```
from sklearn.model_selection import StratifiedKFold
skf = StratifiedKFold(n_splits=20)
skf.get_n_splits(X, y)
acc_score = []
model = Ridge(alpha=500.0, fit_intercept=True, normalize=False, copy_X=True, max_iter=None, tol=0.001, solver='auto', ra
model.fit(X_train, y_train)


print(skf)
StratifiedKFold(n_splits=5, random_state=None, shuffle=False)
for train_index, test_index in skf.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]
    model.fit(X_train,y_train)
    y_pred = model.predict(X_test)
    acc = r2_score(y_test, y_pred)
    acc_score.append(acc)


calc_acc(acc_score)
```

```
StratifiedKFold(n_splits=20, random_state=None, shuffle=False)
accuracies  [0.7123450721149015, 0.6516375857949669, 0.6695310190524733, 0.6588821768601779, 0.6364012464825681, 0.620208467
3888018, 0.6655093460696506, 0.6635682925703837, 0.6487246667335878, 0.6656284566829771, 0.6930242708682257, 0.6883656116085
477, 0.6673137631762613, 0.6821162645101901, 0.7571936824011432, 0.6656587249673309, 0.650740613782004, 0.6851274237603708,
0.6688659735798297, 0.6558637075047213]
Maximum accuracy 75.71936824011432 %  fold # 14
```

```
X:\Dev\Apps\Anaconda3\lib\site-packages\sklearn\model_selection\_split.py:670: UserWarning: The least populated class in y h
as only 1 members, which is less than n_splits=20.
  warnings.warn(("The least populated class in y has only %d"
```

We are using stratified K fold with regularized linear regression without normalization, using alpha=500 and 20 folds. The best accuracy in fold number 14, and best accuracy of 75.7193%

Second part: Implementing it from scratch without using sci-kit library

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
dataset = pd.read_csv('house_prices_data_training_data.csv', sep=',', header=None)
dataset.dropna(inplace=True)
dataset = dataset.drop(0)
dataset = dataset.drop(0, axis = 1)
dataset = dataset.drop(1, axis = 1)

dataset = dataset.convert_dtypes()

dataset.apply(pd.to_numeric).dtypes
dataset = dataset.astype(float)
```

First of all, we are importing pandas, numpy libraries then we will load our uploaded dataset and save it into a variable and we will drop all the rows of missing values. Moreover, we will drop the first 2 columns **“ID”, “Date”** because they are not essential to predict our data. We are also converting the datatype of our data frames into float.

```python
def feature_normalize(X, mean=np.zeros(1), std=np.zeros(1)):
    X = np.array(X)
    if len(mean.shape) == 1 or len(std.shape) == 1:
        mean = np.mean(X, axis=0)
        std = np.std(X, axis=0, ddof=1)

    X = (X - mean)/std
    return X, mean, std
def compute_cost(X, y, theta):
    m = y.shape[0]
    h = X.dot(theta)
    J = (1/(2*m)) * ((h-y).T.dot(h-y))
    return J

def gradient_descent(X, y, theta, alpha, num_iters):
    m = y.shape[0]
    J_history = np.zeros(shape=(num_iters, 1))

    for i in range(0, num_iters):
        h = X.dot(theta)
        diff_hy = h - y

        delta = (1/m) * (diff_hy.T.dot(X))
        theta = theta - (alpha * delta.T)
        J_history[i] = compute_cost(X, y, theta)

    return theta, J_history
```

feature_normalize function, we are doing normalization. We are fetching all the values in the columns then subtract the mean value of this column and divide it by the standard deviation to normalize our data.

Compute_cost function**,** we are calculating the cost to use it in the gradient descent and optimize our model.

Gradient_descent function: In our code, we are trying to solve the cost function, and find its parameters Theta to minimize the cost function. We are using the equation of the gradient descent that does repetition.

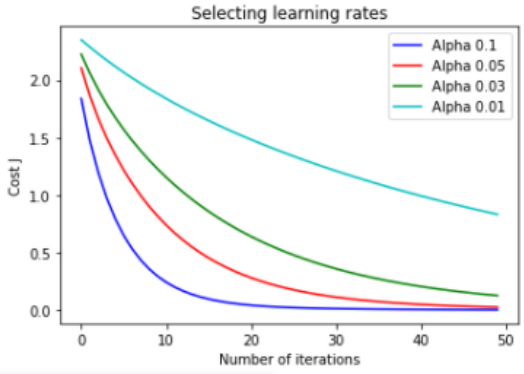
Gradient descent definition: Is an optimization algorithm for finding a local minimum of a differentiable function. It is used to find the values of a function's parameters(coefficients) that minimize a cost function as far as possible.

Cost function definition: The difference between the predicted value(hypothesis) and the actual value. This function used to quantify this loss during the training phase. It used in the supervised learning algorithms that use optimization techniques.

Normalization definition: It is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0 and 1. This scaling helps the gradient descent converge more quickly towards the global minimum.

```
X_norm, mu, sigma = feature_normalize(dataset)
print('mean:', mu)
print('standard deviation:', sigma)
m = dataset.shape[0]
X = np.hstack((np.ones((m,1)),X_norm))
y = np.array(dataset.iloc[:, 2].astype(float).values).reshape(-1,1)
theta = np.zeros(shape=(X.shape[1],1))
alpha = [0.1, 0.05, 0.03, 0.01]
colors = ['b','r','g','c']
num_iters = 50
for i in range(0, len(alpha)):
    theta = np.zeros(shape=(X.shape[1],1))
    theta, J_history = gradient_descent(X, y, theta, alpha[i], num_iters)
    plt.plot(range(len(J_history)), J_history, colors[i], label='Alpha {}'.format(alpha[i]))
plt.xlabel('Number of iterations');
plt.ylabel('Cost J');
plt.title('Selecting learning rates');
plt.legend()
plt.show()
```

```
mean: [ 5.33074250e+05  3.36296461e+00  2.06160064e+00  2.05182232e+03
  1.57587327e+04  1.43316295e+00  7.83376854e-03  2.42457914e-01
  3.44852492e+00  7.59236624e+00  1.74974232e+03  3.02080004e+02
  1.96728235e+03  9.02535141e+01  9.80780698e+04  4.75602109e+01
 -1.22215112e+02  1.97323674e+03  1.32822707e+04]
standard deviation: [3.64412151e+05 9.34032388e-01 7.58632316e-01 9.02744154e+02
 4.24551425e+04 5.09662990e-01 8.81636686e-02 7.78856958e-01
 6.67039405e-01 1.16949482e+00 8.05427363e+02 4.49443008e+02
 2.79403058e+01 4.14717703e+02 5.41249245e+01 1.39644240e-01
 1.39135290e-01 6.73186870e+02 2.73524150e+04]
```



First, we are reading the output of the normalization function and saving it in 3 outputs, X_Norm, M, and Sigma.

We are adding to the X Data frames dummy variable that is equal to ones and multiplied by the bias variable Theta0. Then we are initializing our Theta with zeros with the same length of input column.

Alpha represents the learning rate, and we are trying different values for it to see which one will give the global minimum in the gradient descent. And the number of iterations = 50. This graph represents the curves for different learning rate, and we can notice that the best Alpha is 0.1 because it decreases faster than the other Alphas.

Learning rate(Alpha) definition: It determines how big the gradient descent takes into the direction of the local minimum, and it figures out how fast or slow we will move towards the optimal weight.

```
iterations = 250
alpha = 0.1
theta, j = gradient_descent(X, y, theta, alpha, iterations)
print('Thetas')
print(theta)

bedrooms = (2, - mu[0]/sigma[0])
bathrooms = (1.75, - mu[1]/sigma[1])
sqft_living = (3000, mu[2]/sigma[2])
sqft_lot = (5560, mu[3]/sigma[3])
floors = (1.5, mu[4]/sigma[4])
waterfront = (0, mu[5]/sigma[5])
view = (4, mu[6]/sigma[6])
condition = (5, mu[7]/sigma[7])
grade = (11, mu[8]/sigma[8])
sqft_above = (1300, mu[9]/sigma[9])
sqft_basement = (300, mu[10]/sigma[10])
yr_built = (2004, mu[11]/sigma[11])
yr_renovated = (1800, mu[12]/sigma[12])
zipcode = (98017, mu[13]/sigma[13])
lat = (47.5316, mu[14]/sigma[14])
long_ = (-122.115, mu[15]/sigma[15])
sqft_living15 = (1700, mu[16]/sigma[16])
sqft_lot15 = (3770, mu[17]/sigma[17])

y_pred = theta[0] + theta[1]*bedrooms + theta[2]*bathrooms + theta[3]*sqft_living + theta[4]*sqft_lot + theta[5]*floors + theta[6]*waterfront + theta[7]*view + theta[8]*condition + theta[9]*g
f'Price of a house with 1650 square feet and 3 bedrooms: {y_pred[0]}$'
```

```
Thetas
[[ 2.06160064e+00]
 [ 4.36136659e-04]
 [ 1.13119183e-04]
 [ 7.56835903e-01]
 [ 7.76800922e-04]
 [ 4.49622710e-05]
 [ 3.19657425e-04]
 [-1.31744205e-04]
 [ 4.01519841e-05]
 [ 2.08771402e-04]
 [-7.58462619e-04]
 [ 6.20533146e-04]
 [ 4.48239515e-04]
 [ 1.14614056e-03]
```

We are predicting the price of some customized house, given certain features that are not mentioned.

```
[53] def normal_eqn(X, y):
         inv = np.linalg.pinv(X.T.dot(X))
         theta = inv.dot(X.T).dot(y)
         return theta
```

We are solving the linear regression using the normal equation method.

```
[54] Xe = np.hstack((np.ones((m,1)),X_norm))
     theta_e = normal_eqn(Xe, y)

     bedrooms = (4, - mu[0]/sigma[0])
     bathrooms = (1.25, - mu[1]/sigma[1])
     sqft_living = (1420, mu[2]/sigma[2])
     sqft_lot = (6230, mu[3]/sigma[3])
     floors = (3, mu[4]/sigma[4])
     waterfront = (0, mu[5]/sigma[5])
     view = (2, mu[6]/sigma[6])
     condition = (4, mu[7]/sigma[7])
     grade = (8, mu[8]/sigma[8])
     sqft_above = (880, mu[9]/sigma[9])
     sqft_basement = (590, mu[10]/sigma[10])
     yr_built = (1940, mu[11]/sigma[11])
     yr_renovated = (1880, mu[12]/sigma[12])
     zipcode = (98019, mu[13]/sigma[13])
     lat = (47.7088, mu[14]/sigma[14])
     long_ = (-122.045, mu[15]/sigma[15])
     sqft_living15 = (1800, mu[16]/sigma[16])
     sqft_lot15 = (6730, mu[17]/sigma[17])

     y_pred = theta_e[0] + theta_e[1]*bedrooms + theta_e[2]*bathrooms + theta_e[3]*sqft_living + theta_e[4]*sqft_lot + theta_e[5]*floors + theta_e[6]*waterfront +
     f'Price of a house with 1650 square feet and 3 bedrooms: {y_pred[0]}$'

     'Price of a house with 1650 square feet and 3 bedrooms: 1079.3194891348307$'
```

We are outputting the theta from the normal equation to use it and predict the price of a custom house.

```
[55] dataset = pd.read_csv('house_prices_data_training_data.csv', sep=',')
     dataset.dropna(inplace=True)
     dataset = dataset.drop(columns='id', axis = 1)
     dataset = dataset.drop(columns='date', axis = 1)

     dataset = dataset.convert_dtypes()

     dataset = dataset.astype(float)
     y = dataset.iloc[:, 1]
     X = dataset.iloc[:, 2:]
     X["bathrooms1"] = X["bathrooms"]**2
     X["floors1"] = X["floors"]**2
     X["sqft_living1"] = X["sqft_living"]**2
     X["sqft_living2"] = X["sqft_living"]**3
     X["sqft_lot151"] = X["sqft_lot15"]**4
     X["condition1"] = X["condition"]**2
     X["zipcode1"] = X["zipcode"]**3
```

We are re-initializing my dataset to use in the polynomial regression. We are also dropping the empty rows and dropping the “ID” and “Date” columns that are not important to predict our output.

We are using a polynomial of degree 4 in my implementation.

```
df = pd.read_csv('house_prices_data_training_data.csv', sep=',')

def split_train_valid_test(data,valid_ratio,test_ratio):
    shuffled_indcies=np.random.permutation(len(data))
    valid_set_size= int(len(data)*valid_ratio)
    valid_indcies=shuffled_indcies[:valid_set_size]
    test_set_size= int(len(data)*test_ratio)
    test_indcies=shuffled_indcies[valid_set_size:test_set_size+valid_set_size]
    train_indices=shuffled_indcies[test_set_size:]
    return data.iloc[train_indices],data.iloc[valid_indcies],data.iloc[test_indcies]

train_set,valid_set,test_set=split_train_valid_test(dataset,valid_ratio=0.2,test_ratio=0.2)
```

We are splitting our data into test dataset, training dataset and cross-validation data set using the model selection.

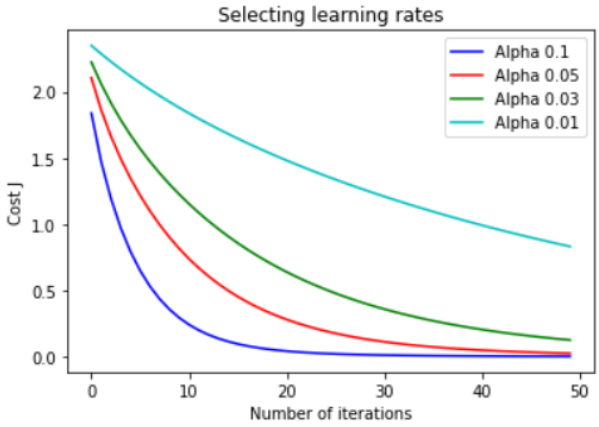
```
y = train_set.iloc[:, 1]
X = train_set.iloc[:, 2:]
X["bathrooms1"] = X["bathrooms"]**2
X["floors1"] = X["floors"]**2
X["sqft_living1"] = X["sqft_living"]**2
X["sqft_living2"] = X["sqft_living"]**3
X["sqft_lot151"] = X["sqft_lot15"]**4
X["condition1"] = X["condition"]**2
X["zipcode1"] = X["zipcode"]**3
X_norm, mu, sigma = feature_normalize(dataset)
print('mean:', mu)
print('standard deviation:', sigma)
m = dataset.shape[0]
X = np.hstack((np.ones((m,1)),X_norm))

y = np.array(dataset.iloc[:, 2].astype(float).values).reshape(-1,1)
theta = np.zeros(shape=(X.shape[1],1))
alpha = [0.1, 0.05, 0.03, 0.01]
colors = ['b','r','g','c']
num_iters = 50
X.shape
```

```
mean: [ 5.33074250e+05  3.36296461e+00  2.06160064e+00  2.05182232e+03
  1.57587327e+04  1.43316295e+00  7.83376854e-03  2.42457914e-01
  3.44852492e+00  7.59236624e+00  1.74974232e+03  3.02080004e+02
  1.96728235e+03  9.02535141e+01  9.80780698e+04  4.75602109e+01
 -1.22215112e+02  1.97323674e+03  1.32822707e+04]
standard deviation: [3.64412151e+05 9.34032388e-01 7.58632316e-01 9.02744154e+02
 4.24551425e+04 5.09662990e-01 8.81636686e-02 7.78856958e-01
 6.67039405e-01 1.16949482e+00 8.05427363e+02 4.49443008e+02
 2.79403058e+01 4.14717703e+02 5.41249245e+01 1.39644240e-01
 1.39135290e-01 6.73186870e+02 2.73524150e+04]
```

We are using different polynomial models of degree 4 in our training dataset.

```
[60] for i in range(0, len(alpha)):
         theta = np.zeros(shape=(X.shape[1],1))
         theta, J_history = gradient_descent(X, y, theta, alpha[i], num_iters)
         plt.plot(range(len(J_history)), J_history, colors[i], label='Alpha {}'.format(alpha[i]))
     plt.xlabel('Number of iterations');
     plt.ylabel('Cost J');
     plt.title('Selecting learning rates');
     plt.legend()
     plt.show()
```



The best learning rate in this case is 0.1.