Information Engineering and Technology Faculty
German University in Cairo

NETW 1013: Machine Learning

Assignment 1 Report

Author: Fatma Salaheldeen Helmy Abdelkhaleq Abothekry

Supervisor: Dr. Maggie Ezzat

Linear Regression Report

In this report, the results of the three techniques applied in assignment 1 will be represented and compared along with explained snippets for the code.

1) Features Selection

Before applying any techniques, the most important features that affect the output “price” need only to be used not the entire dataset. A heatmap was generated, as seen in figure [1], to investigate the correlation between all features and output. It was generated using the code in figure [2], but the “id” column was dropped because it’s not a feature of the house.

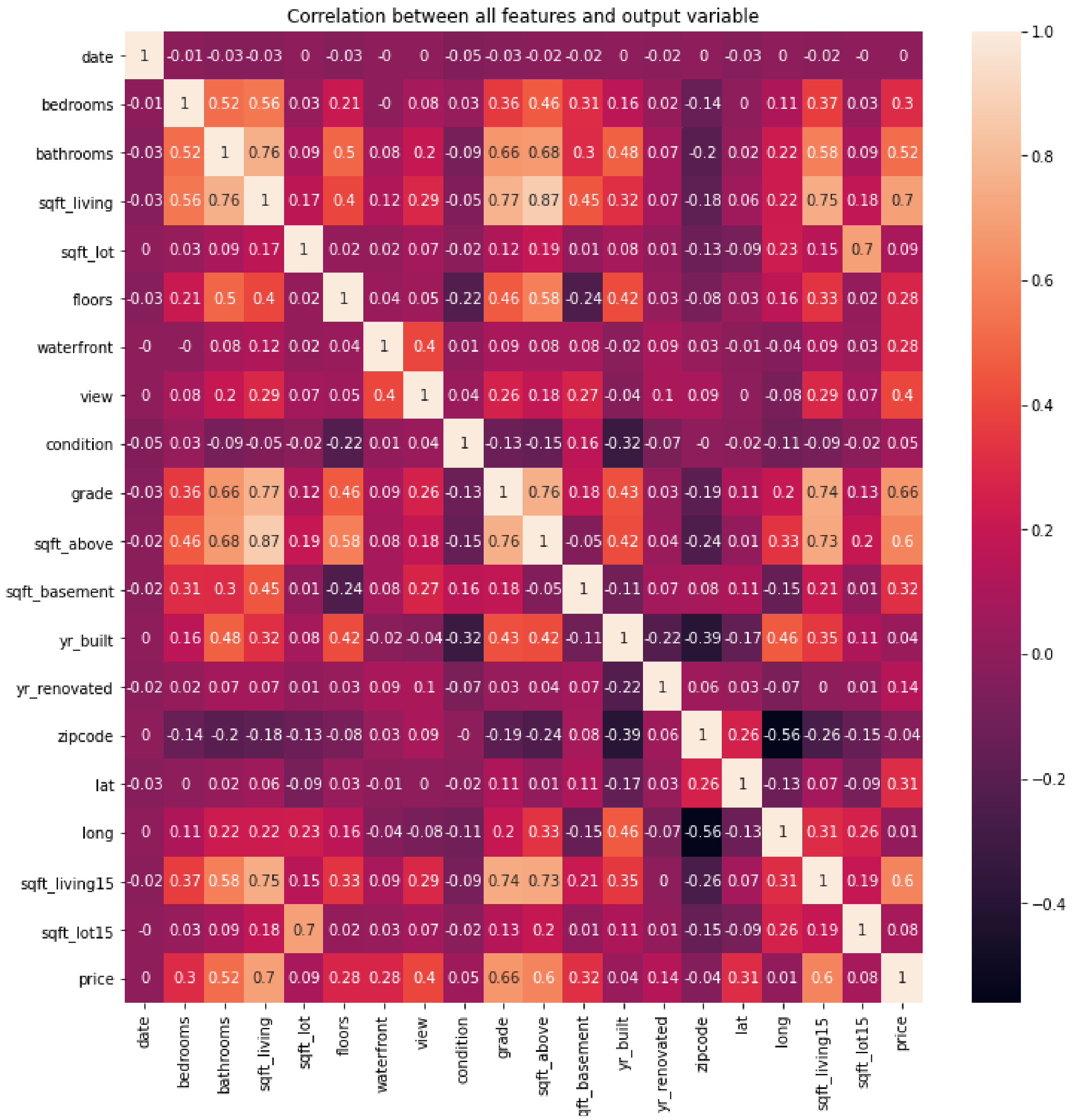
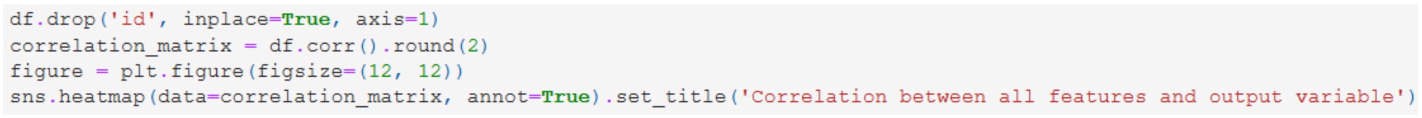


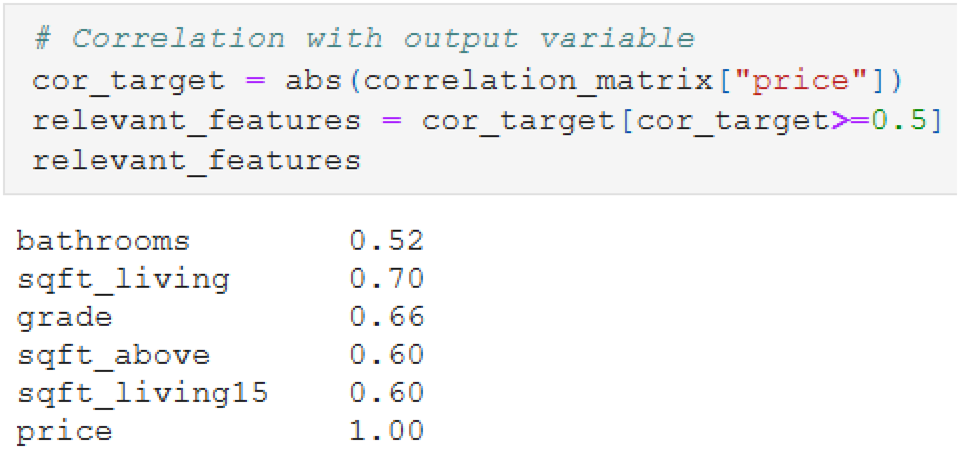
Figure 1: Heatmap



```
df.drop('id', inplace=True, axis=1)
correlation_matrix = df.corr().round(2)
figure = plt.figure(figsize=(12, 12))
sns.heatmap(data=correlation_matrix, annot=True).set_title('Correlation between all features and output variable')
```

Figure 2: Heatmap generation code

The heatmap shows that only a few features have an absolute correlation with the "price" greater than 0.5. These features are "bathrooms", "sqft_living", "grade", "sqft_above", and "sqft_living15" as shown in figure [3]. Therefore, only these features will be used in Model Selection and K-fold techniques but not in the Regularization technique because it mainly depends on keeping all features.



```
# Correlation with output variable
cor_target = abs(correlation_matrix["price"])
relevant_features = cor_target[cor_target>=0.5]
relevant_features
```

```
bathrooms        0.52
sqft_living      0.70
grade            0.66
sqft_above       0.60
sqft_living15    0.60
price            1.00
```

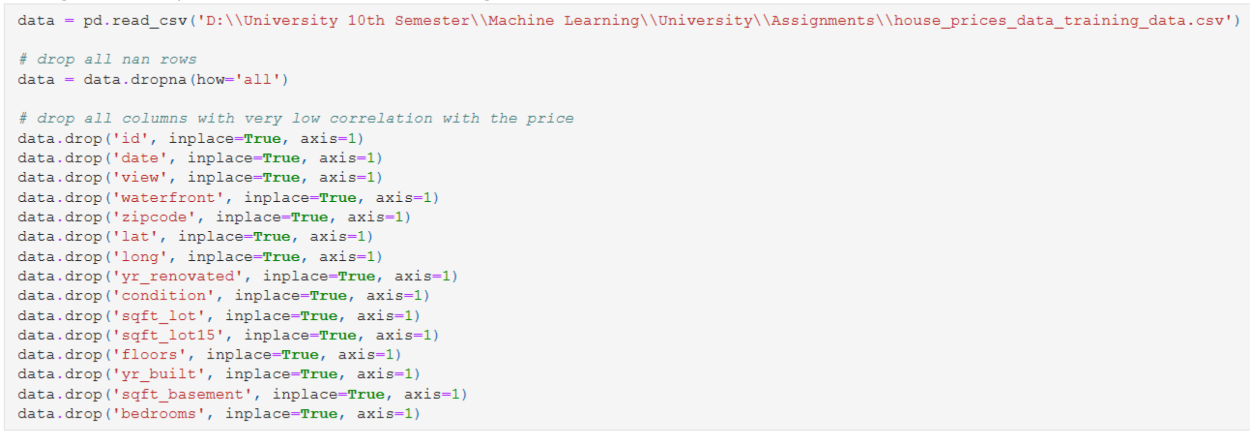
Figure 3: Relevant features to the output

2) Model Selection

In this technique 10 hypothesis functions with polynomial degrees ranging from 1 to 10 will be investigated in order to choose the degree that best fits the features selected in section 1.

a) Read the Data

First, the dataset is read, and the features that weren't selected in section 1 are dropped along with any null rows as seen in figure [4].

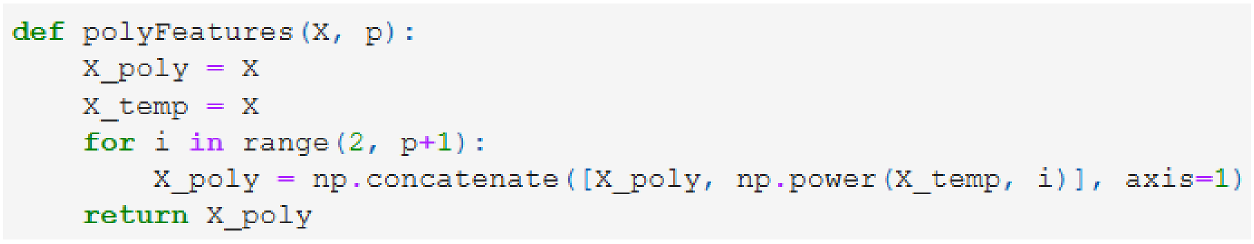


```
data = pd.read_csv('D:\\University 10th Semester\\Machine Learning\\University\\Assignments\\house_prices_data_training_data.csv')

# drop all nan rows
data = data.dropna(how='all')

# drop all columns with very low correlation with the price
data.drop('id', inplace=True, axis=1)
data.drop('date', inplace=True, axis=1)
data.drop('view', inplace=True, axis=1)
data.drop('waterfront', inplace=True, axis=1)
data.drop('zipcode', inplace=True, axis=1)
data.drop('lat', inplace=True, axis=1)
data.drop('long', inplace=True, axis=1)
data.drop('yr_renovated', inplace=True, axis=1)
data.drop('condition', inplace=True, axis=1)
data.drop('sqft_lot', inplace=True, axis=1)
data.drop('sqft_lot15', inplace=True, axis=1)
data.drop('floors', inplace=True, axis=1)
data.drop('yr_built', inplace=True, axis=1)
data.drop('sqft_basement', inplace=True, axis=1)
data.drop('bedrooms', inplace=True, axis=1)
```

Figure 4: Data reading

b) Functions

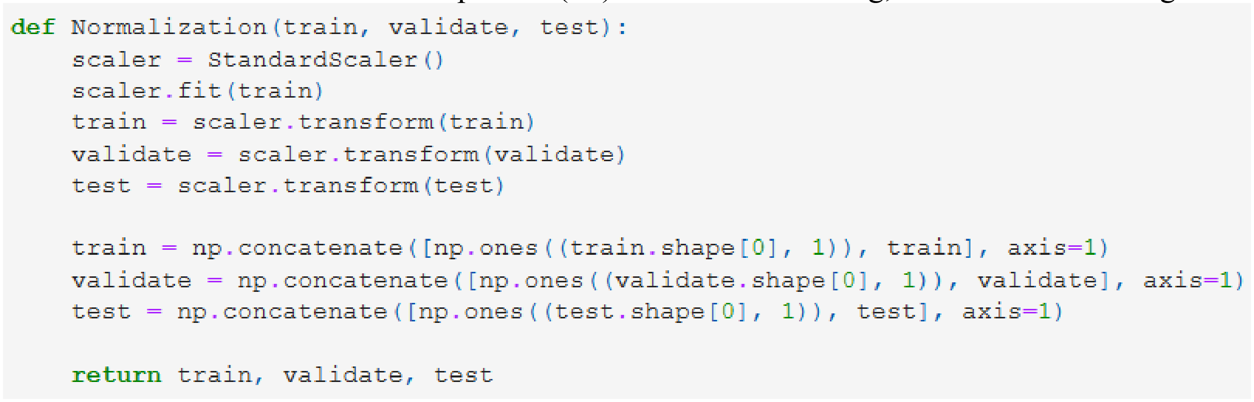
Four functions are used in Model Selection. The first function "polyFeatures" in figure [5] takes as input a 2-D array and a number. This number represents the desired degree of the hypothesis function. It raises all array's features to power 2 and all subsequent powers until the desired power 'p'. Then, it returns another array that has all powers of the array concatenated with each other starting from power 1 until power 'p'. Like this each feature has 'p' columns where each column represents a single power of the feature. This means that an input array with 'm' columns will result in an output array of 'm*p' columns.



```
def polyFeatures(X, p):
    X_poly = X
    X_temp = X
    for i in range(2, p+1):
        X_poly = np.concatenate([X_poly, np.power(X_temp, i)], axis=1)
    return X_poly
```

Figure 5: polyFeatures function

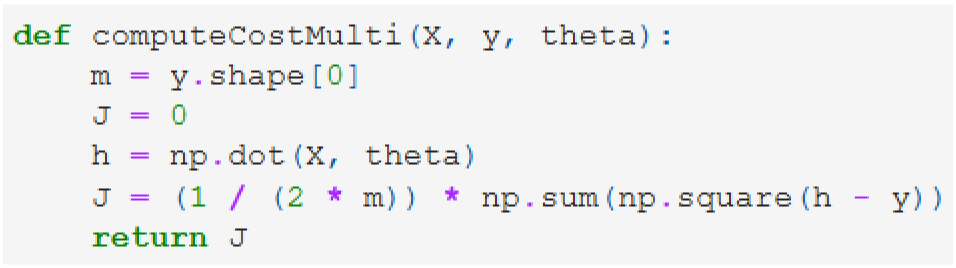
The second function "Normalization" in figure [6] takes as input the features of the train, validation and test sets that are already raised to the power equal to the hypothesis function's degree and outputs the three sets but with mean normalization performed (mean = 0 and standard deviation = 1). It uses a StandardScaler defined by sklearn that fits the train set then normalizes/transforms the three sets. After normalization, it adds to the beginning of each set a column of ones which is the intercept term (X_0) used later in training, validation and testing.



```
def Normalization(train, validate, test):
    scaler = StandardScaler()
    scaler.fit(train)
    train = scaler.transform(train)
    validate = scaler.transform(validate)
    test = scaler.transform(test)

    train = np.concatenate([np.ones((train.shape[0], 1)), train], axis=1)
    validate = np.concatenate([np.ones((validate.shape[0], 1)), validate], axis=1)
    test = np.concatenate([np.ones((test.shape[0], 1)), test], axis=1)

    return train, validate, test
```

Figure 6: Normalization function

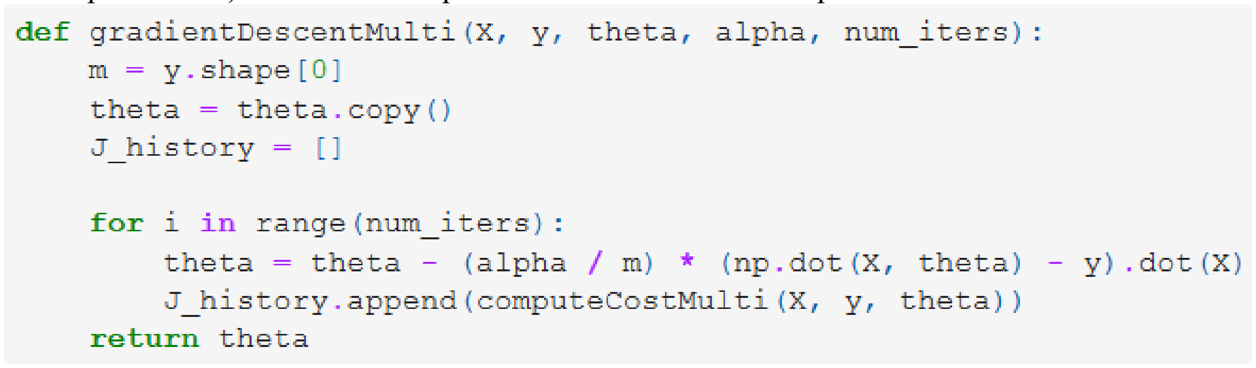
The third function "computeCostMulti" in figure [7] takes as input the train/test/validation input set, train/test/validation output set and thetas array in order to calculate (J) which is the cost function of linear regression. **The input set is all features' columns, and the output set is the "price" column that should be predicted.**



```python
def computeCostMulti(X, y, theta):
    m = y.shape[0]
    J = 0
    h = np.dot(X, theta)
    J = (1 / (2 * m)) * np.sum(np.square(h - y))
    return J
```

Figure 7: computeCostMulti Function

Lastly, function “gradientDescentMulti” in figure [8] takes as input the train input set, train output set, thetas array, alpha (learning rate) and number of iterations. It keeps on iterating and calculating the cost until convergence (reaching the minimum cost and optimized values of thetas/parameters). It returns the optimized values of the thetas/parameters.

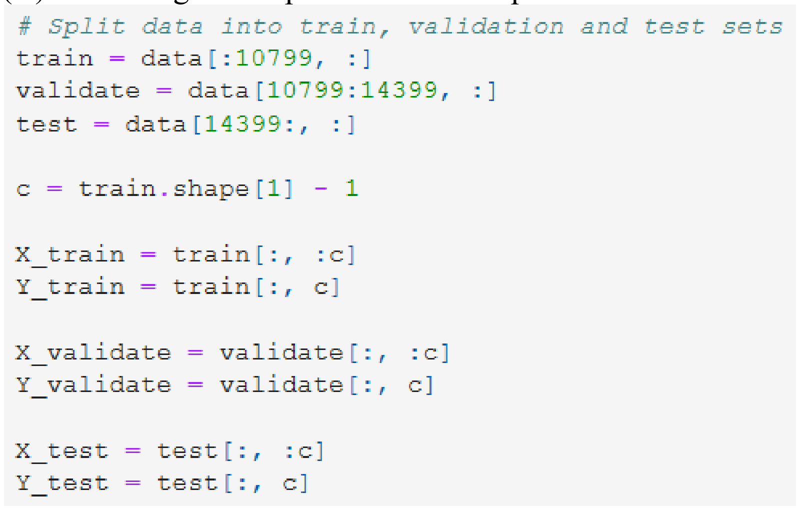


```python
def gradientDescentMulti(X, y, theta, alpha, num_iters):
    m = y.shape[0]
    theta = theta.copy()
    J_history = []

    for i in range(num_iters):
        theta = theta - (alpha / m) * (np.dot(X, theta) - y).dot(X)
        J_history.append(computeCostMulti(X, y, theta))
    return theta
```

Figure 8: gradientDescentMulti Function

c) Prepare the Data

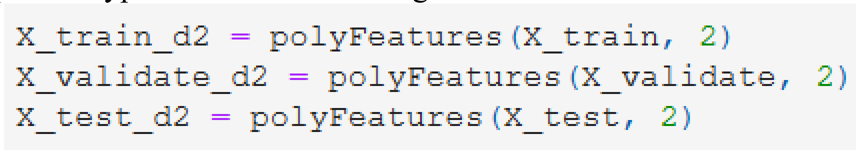
Before training, validating and testing, the dataset needs to be prepared. First, it’s split into three sets which are the train, validation and test sets as seen in figure [9]. The train set represents 60 % of the dataset while the validation and test sets represent 40 % of the dataset. Each of them represents 20 % of the dataset. Then each set is divided into an input set (X) containing all features and an output set (Y) containing the output which is the “price” column.



```python
# Split data into train, validation and test sets
train = data[:10799, :]
validate = data[10799:14399, :]
test = data[14399:, :]

c = train.shape[1] - 1

X_train = train[:, :c]
Y_train = train[:, c]

X_validate = validate[:, :c]
Y_validate = validate[:, c]

X_test = test[:, :c]
Y_test = test[:, c]
```

Figure 9: Dataset splitting

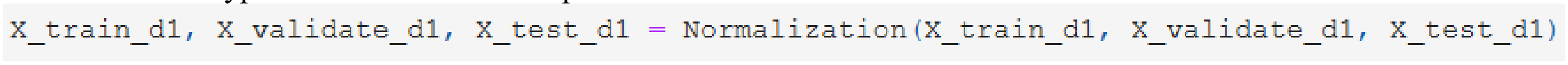
Second, train, validation and test input sets of power 1 are raised to powers ranging from 2 to 10. This is done to later use these sets with hypothesis functions of degrees ranging from 2 to 10. It's done in figure [10] by using "polyFeatures" function which will return the three sets suitable for a specific hypothesis function of degree 'd'.



```
X_train_d2 = polyFeatures(X_train, 2)
X_validate_d2 = polyFeatures(X_validate, 2)
X_test_d2 = polyFeatures(X_test, 2)
```

Figure 10: Sets power calculation

Lastly, all train, validation and test sets of all powers are normalized for two reasons. First, to make all features on the same scale so that the gradient descent converges faster. Second, to avoid overflow in running. As seen in figure [11], the "Normalization" function is used and takes as input the train, validation and test input sets of power 'd' which corresponds to the degree of the hypothesis function and outputs the same sets but normalized.

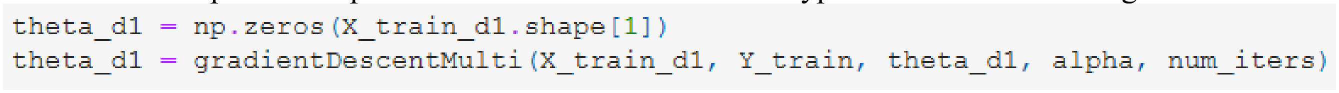


```
X_train_d1, X_validate_d1, X_test_d1 = Normalization(X_train_d1, X_validate_d1, X_test_d1)
```

Figure 11: Sets normalization

d) Train

Ten arrays are created for the thetas of the 10 hypothesis functions of degrees 1 to 10. Each array represents thetas of a hypothesis function of degree 'd' and has size equal to the number of features/columns in the training input set of degree 'd' as seen in figure [12]. Then, each thetas array for hypothesis function of degree 'd' is obtained by using the "gradientDescentMulti" function which takes as input the thetas array, train input set of degree 'd', train output set, learning rate = 0.01 and number of iterations = 400. The same function is used another 9 times with the other train input sets of powers 2 to 10 to obtain thetas of hypothesis functions of degrees 2 to 10.

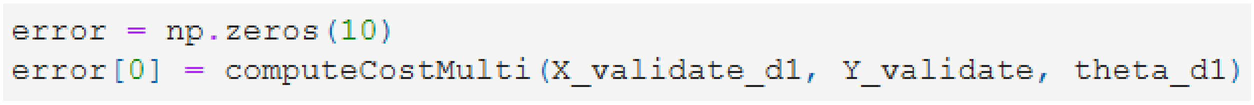


```
theta_d1 = np.zeros(X_train_d1.shape[1])
theta_d1 = gradientDescentMulti(X_train_d1, Y_train, theta_d1, alpha, num_iters)
```

Figure 12: Thetas calculation

e) Cross Validation Error

After calculating all thetas of all hypothesis functions, the validation error of each hypothesis function needs to be calculated. In figure [13], an error array of size 10 is created to save the validation errors of the 10 hypothesis functions of degrees 1 to 10. To calculate this error, the "computeCostMulti" function is used which takes as input the obtained thetas array of hypothesis function of degree 'd', validation input set of degree 'd' and validation output set. The same function is used another nine times on the other validation input sets of powers 2 to 10 and their corresponding thetas arrays to obtain the remaining validation errors.



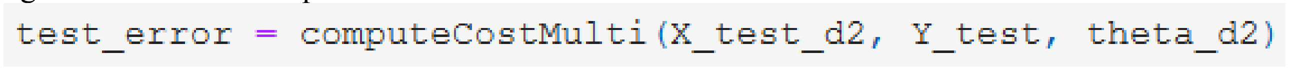
```
error = np.zeros(10)
error[0] = computeCostMulti(X_validate_d1, Y_validate, theta_d1)
```

Figure 13: Cross validation error

After calculating the ten validation errors for the ten hypothesis functions, we need to know which hypothesis function achieves the least validation error in order to calculate its generalization error. The obtained cross validation errors are [2.89671149e+10, 2.88192726e+10, 3.66046707e+10, 4.15499546e+10, 4.26038702e+10, 4.14967603e+10, 3.95742116e+10, 3.75725227e+10, 3.57274746e+10, 3.40402347e+10] which indicates that the hypothesis function of degree 2 has the least cross validation error.

f) Generalization Error

Lastly, the test input set of power 2 and test output set are used to calculate the generalization error of the hypothesis of degree 2 to see how well it performs on data it hasn't seen before. To calculate the error, "computeCostMulti" function is used in figure [14] and takes as input the thetas array of hypothesis function of degree 2, test output set and test input set. It gives generalization error equal to 25310675301.093674.



```
test_error = computeCostMulti(X_test_d2, Y_test, theta_d2)
```

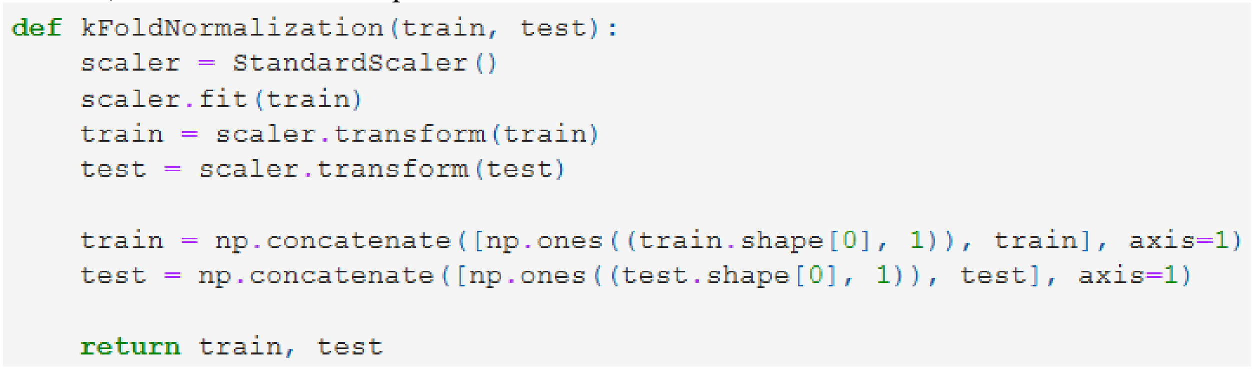
Figure 14: Generalization error

3) K-fold Sampling

In this technique 5 hypothesis functions of degrees ranging from 1 to 5 will be investigated in order to choose the polynomial degree that best fits the features chosen in section 1. The difference between this technique and the previous one is that we don't train and test the hypothesis function just once, but we do this several times in order to get the mean generalization error.

a) Functions

The same functions are used as in Model Selection except for the "Normalization" function. Here another function called "kFoldNormalization" is used. In figure [15] it does exactly the same thing as the "Normalization" function but using only the train and test input sets and not the train, validation and test input sets.



```
def kFoldNormalization(train, test):
    scaler = StandardScaler()
    scaler.fit(train)
    train = scaler.transform(train)
    test = scaler.transform(test)

    train = np.concatenate([np.ones((train.shape[0], 1)), train], axis=1)
    test = np.concatenate([np.ones((test.shape[0], 1)), test], axis=1)

    return train, test
```

Figure 15: kFoldNormalization Function

b) Prepare the Folds, Train and Test Sets

In this part KFold defined by sklearn is used to create 5 folds. In each fold 1/5 of the dataset is a test set and the other 4/5 is a train set. In the first fold the test set is the first 1/5 rows of the dataset, and the remaining rows are the train set. In the second fold the test set is the second 1/5 rows of the dataset, and the remaining rows are the train set. In the third fold the test set is the third 1/5 rows of the dataset, and the remaining rows are the train set. In the fourth fold the test set is the fourth 1/5 rows of the dataset, and the remaining rows are the train set. In the fifth fold the test set is the last 1/5 rows of the dataset, and the remaining rows are the train set.

In figure [16] a for loop is created in order to create 5 different train and test sets. It's run for 5 times where 'i' represents the fold number. For each fold, the train and test input sets are saved in 2 different variables. Also, the train and test output sets are saved in 2 different variables. Like this we have 5 different train input, test input, train output and test output sets.

```python
kf = KFold(n_splits=5)
i = 1
for train_index, test_index in kf.split(data):
    if i == 1:
        x_train_d1_1, x_test_d1_1 = data[train_index, :c], data[test_index, :c]
        y_train_1, y_test_1 = data[train_index, c], data[test_index, c]

    elif i == 2:
        x_train_d1_2, x_test_d1_2 = data[train_index, :c], data[test_index, :c]
        y_train_2, y_test_2 = data[train_index, c], data[test_index, c]

    elif i == 3:
        x_train_d1_3, x_test_d1_3 = data[train_index, :c], data[test_index, :c]
        y_train_3, y_test_3 = data[train_index, c], data[test_index, c]

    elif i == 4:
        x_train_d1_4, x_test_d1_4 = data[train_index, :c], data[test_index, :c]
        y_train_4, y_test_4 = data[train_index, c], data[test_index, c]

    elif i == 5:
        x_train_d1_5, x_test_d1_5 = data[train_index, :c], data[test_index, :c]
        y_train_5, y_test_5 = data[train_index, c], data[test_index, c]

    i = i + 1
```

Figure 16: Folds creation

Then, the 5 different train and test input sets are raised to powers ranging from 2 to 5. This is done to later use these sets with hypothesis functions of degrees ranging from 2 to 5. As we've done before in Model Selection, in figure [17] "polyFeatures" function is used and returns the two sets suitable for a specific hypothesis function of degree 'd'.

```python
x_train_d2_1 = polyFeatures(x_train_d1_1, 2)
x_test_d2_1 = polyFeatures(x_test_d1_1, 2)
```

Figure 17: Sets power calculation

Lastly, after calculating all the powers of the 5 different train and test input sets, all the sets are normalized using the "kFoldNormalization" function as seen in figure [18]. It takes as input the train and test input sets of fold 'i' and degree 'd' and returns the normalized versions of them. This means that each degree has 5 different normalized train and test input sets and 5 different train and test output sets but not normalized.

```
x_train_d1_1, x_test_d1_1 = kFoldNormalization(x_train_d1_1, x_test_d1_1)
```

Figure 18: Sets normalization

c) Train

In figure [19], 25 arrays are created for the thetas of the 5 folds of the 5 hypothesis functions of degrees 1 to 5. Each array represents thetas of a hypothesis function of degree 'd' and fold 'i'. It has size equal to the number of features/columns in the train input set of degree 'd' and fold 'i'.

Then, each thetas array for hypothesis function of degree 'd' and fold 'i' is obtained by using the "gradientDescentMulti" function taking as input the thetas array, train input set of degree 'd' and fold 'i', train output set of fold 'i', learning rate = 0.01 and number of iterations = 400. The function is used 25 times with train input sets of powers 1 to 5 and folds 1 to 5 along with their corresponding train output sets to obtain all thetas of hypothesis functions of degrees 1 to 5.

```
theta_d1_1 = np.zeros(x_train_d1_1.shape[1])
theta_d1_1 = gradientDescentMulti(x_train_d1_1, y_train_1, theta_d1_1, alpha, num_iters)
```

Figure 19: Thetas calculation

d) Test

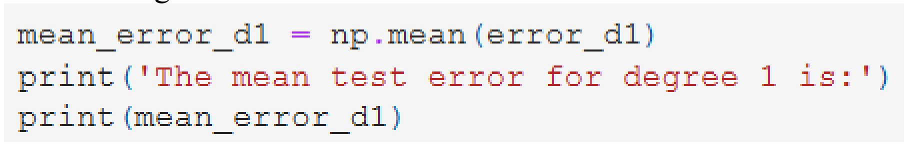
After calculating all thetas of all hypothesis functions of all degrees and folds, the test error of each hypothesis function and fold needs to be calculated. In figure [20], 5 error arrays of size 5 are created to save the test errors of the 5 hypothesis functions of degrees from 1 to 5 and folds 1 to 5. A single array saves the 5 different errors of the 5 folds for a hypothesis function of degree 'd'. To calculate this error, the "computeCostMulti" function is used which takes as input the thetas array of hypothesis function of degree 'd' and fold 'i', test output set of fold 'i' and test input set of degree 'd' and fold 'i'. The function is used 25 times with thetas of powers 1 to 5 and folds 1 to 5 along with their corresponding test output set.

```
error_d1 = np.zeros(5)
error_d1[0] = computeCostMulti(x_test_d1_1, y_test_1, theta_d1_1)
```

Figure 20: Test error

e) Mean Test Error

After calculating the five test errors for each hypothesis function of degree 'd', we need to know which hypothesis function achieves the least mean test error. To do this the mean of every error array is calculated as seen in figure [21]. The obtained mean errors are 31759824698.21289 for degree 1, 27814620131.243126 for degree 2, 28427820743.859325 for degree 3, 29573992797.546173 for degree 4 and 29818165794.40282 for degree 5. This indicates that the hypothesis function of degree 2 has the least mean test error.



```
mean_error_d1 = np.mean(error_d1)
print('The mean test error for degree 1 is:')
print(mean_error_d1)
```

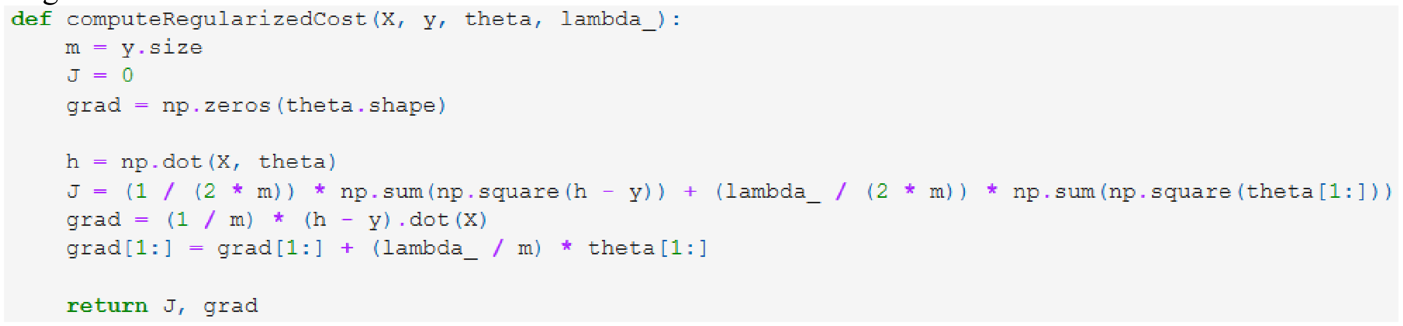
Figure 21: Mean test error

4) Regularization

In this technique 5 hypothesis functions with polynomial degrees ranging from 1 to 5 will be investigated in order to find the simplest hypothesis function with least degree that best fits all features (except the 'id') and not just selected features as in the previous 2 techniques.

a) Functions

"Normalization" and "polyFeatures" functions are used as in Model Selection. As for cost calculation, another function called "computeRegularizedCost" is used as seen in figure [22]. It takes as input the train/test/validation input set, train/test/validation output set, thetas array and lambda (regularization factor) in order to calculate (J) which is the cost function of regularized linear regression. It uses a cost function having a Regularization Term. It outputs (J) and the gradient.



```
def computeRegularizedCost(X, y, theta, lambda_):
    m = y.size
    J = 0
    grad = np.zeros(theta.shape)

    h = np.dot(X, theta)
    J = (1 / (2 * m)) * np.sum(np.square(h - y)) + (lambda_ / (2 * m)) * np.sum(np.square(theta[1:]))
    grad = (1 / m) * (h - y).dot(X)
    grad[1:] = grad[1:] + (lambda_ / m) * theta[1:]

    return J, grad
```

Figure 22: computeRegularizedCost Function

b) Prepare the Data

Preparing the data in Regularization technique has exactly the same steps as the Model Selection technique.

c) Train

Thirty arrays are created for the thetas of the 5 hypothesis functions of degrees 1 to 5 and lambdas = [0, 0.01, 0.02, 0.04, 0.08, 0.16]. Each array represents thetas of a hypothesis function of degree 'd' and lambda 'l'. It has size equal to the number of features/columns in the training input set of degree 'd' as seen in figure [23]. Then, each thetas array for hypothesis function of degree 'd' and lambda 'l' is obtained by using the "trainLinearReg" function offered by utils library. It takes as input the train input set of degree 'd', train output set, lambda and "computeRegularizedCost" function. It performs optimization and outputs the minimized thetas array. The function is used 30 times with train input sets of powers 1 to 5 and lambdas from 0 to 0.16 to obtain thetas of all hypothesis functions and lambdas.

```
theta_d1_lambda_1 = np.zeros(X_train_d1.shape[1])
theta_d1_lambda_1 = utils.trainLinearReg(computeRegularizedCost, X_train_d1, Y_train, lambda_=0)
```

Figure 23: Thetas calculation

d) Cross Validation Error

After calculating all thetas of all hypothesis functions and lambdas, the validation error of each hypothesis function and lambda needs to be calculated. In figure [24], 5 error arrays of size 6 are created to save the validation errors of the 5 hypothesis functions of degrees from 1 to 5 and lambdas from 0 to 0.16. A single array saves the 6 different errors of the 6 lambdas for a hypothesis function of degree 'd'. To calculate this error, the "computeRegularizedCost" function is used which takes as input the obtained thetas array of hypothesis function of degree 'd' and lambda 'l', validation input set of degree 'd', validation output set and lambda = 0. The function is used 30 times on the validation input sets of powers 1 to 5 and their corresponding thetas arrays obtained using lambdas from 0 to 0.16.

```
error_d1 = np.zeros(6)
error_d1[0] = computeRegularizedCost(X_validate_d1, Y_validate, theta_d1_lambda_1, 0)[0]
```

Figure 24: Cross validation error

Lastly, the minimum cost in each error array is obtained as seen in figure [25] by printing the index of minimum error in each array and printing the error corresponding to this index. The index is used to know the position of the error which accordingly tells us the lambda used. The achieved validation errors are 18625940003.498924 (lambda = 0.16), 22961758939.599865 (lambda = 0), 27571976248.760567 (lambda = 0.08), 25756478525.251137 (lambda = 0.01) and 23309595256.42787 (lambda = 0).

```
print('Minimum cost in degree 1 is at index:')
print(np.argmin(error_d1))

print('Minimum test error in degree 1 is:')
print(error_d1[np.argmin(error_d1)])
```

Figure 25: Generalization error

e) Generalization Error

Lastly, the test input set of power 1 and test output set are used to calculate the generalization error of the hypothesis function of degree 1 to see how well it performs on data it hasn't seen before. To calculate the error, "computeRegularizedCost" function is used in figure [26] and takes as input the thetas array of hypothesis function of degree 1 and lambda = 0.16, test output set and test input set. It gives generalization error equal to 25310675301.093674.

```
test_reg_error = computeRegularizedCost(X_test_d1, Y_test, theta_d1_lambda_6, 0.16)[0]
```

Figure 26: Generalization error

5) Techniques Comparison

Lastly, the performance of the 3 techniques is compared. It can be seen that Model Selection resulted in selecting a second-degree hypothesis function. Then, K-fold Sampling chose the same degree but has higher generalization error than Model Selection since it takes the average of the error. Although the error is higher, it gives a more realistic and accurate error than Model Selection. Therefore, this is considered as an enhancement. Then comes the Regularization technique which selects a first-degree hypothesis function. It also shows less generalization error than the other 2 techniques. The improvement here isn't just in achieving lower generalization error but is also in using a simpler hypothesis function that fits the data. Therefore, Regularization has the best performance among the 3 techniques.

	Degree	**Generalization Error**
Model Selection	2	25310675301.093674
K-fold Sampling	2	27814620131.243126
Regularization	1	17799790840.643875