COMP-551: Applied Machine Learning
Programming Assignment #1
Mandana Samiei
Student ID: 260779555

1- Model Selection

1)
We are supposed to fit a 20-degree polynomial to the data:
a 20-degree polynomial is as the following equation, so we need to calculate the matrix of X which is included differet power of feature x from 0 to 20:
predicted_y(x,w) = $w_0x^0 + w_1x^1 + w_2x^2+w_3x^3+w_4x^4+....+w_{20}x^{20} = \sum_{j=0}^{M} w_j x^j$

$$\mathrm{MSE}(w) = 1/N \sum_{i=1}^{N} (y_i - x_i^T w)^2$$

matrix notation of MSE(w) = $(y - Xw)^T * (y - Xw)$

We aim to find the most optimal parameters to make the mean square error as smallest as possible,
So we differentiate MSE with regards to w:

differentiating result: $X^T * (y - Xw) = 0$
$w = X^T y / X^T X = ((X^T X)^{-1} X^T y)$

In this example we have 50 instances so X has 50 rows and 21 column of features: X: 50*21

Y: 50*1
X^T: 21*50 → XX^T: 21*21 →$(XX^T)^{-1}$: 21*21 → $(XX^T)^{-1}X^T$: 21*50 → $w = (XX^T)^{-1}X^T y$: 21*1

```
Polynomial Parameters(W)[ -3.90941121e+00  -1.54809405e+01   1.55837132e+02   1.90995729e+03
  -7.74227221e+03  -3.57478317e+04   1.27144652e+05   2.90170194e+05
  -1.00155749e+06  -1.25476446e+06   4.38805271e+06   3.16203454e+06
  -1.14572060e+07  -4.78922630e+06   1.82417152e+07   4.28711259e+06
  -1.73639042e+07  -2.08485648e+06   9.07757576e+06   4.23245419e+05
  -2.00440032e+06]
```

A part of code corresponds to close form training:

```
def train_close(self):
    self.feat = self.get_poly(self.X)
    self.w = np.dot(np.dot(inv(np.dot(self.feat.T, self.feat)), self.feat.T), self.Y)
    print "Polynomial Parameters(W):\n{}".format(self.w)
```

Train MSE: 6.47470396778, **Valid MSE:** 1417.89812218, **Test MSE:** 50.653704407

```
def compute_loss(self,degree=20):
    train_mse = ((np.dot(self.feat, self.w) - self.Y)**2).mean(axis=0)
    valid_mse = ((np.dot(self.get_poly(self.valid_x,degree), self.w) - self.valid_y)**2).mean(axis=0)
    test_mse = ((np.dot(self.get_poly(self.test_x,degree), self.w) - self.test_y)**2).mean(axis=0)
    print "Train MSE:{}, Valid MSE:{}, Test MSE:{}".format(train_mse, valid_mse, test_mse)
```

In the next part we are asked to plot the fit with regards to train, validation and test data points:

According to the figure 1 we can see that the model is pruned to overfitting! At some cases, the diagram crosses the training data points, so that it is very specific to the rain data. While our goal is to find a general model of data to be able to predict new data points. Furthermore, in some points we have outliers that indicate the model is very sensitive to the data points.
So we are going to use regularization to remove overfitting phenomenon and reduce the parameter values.

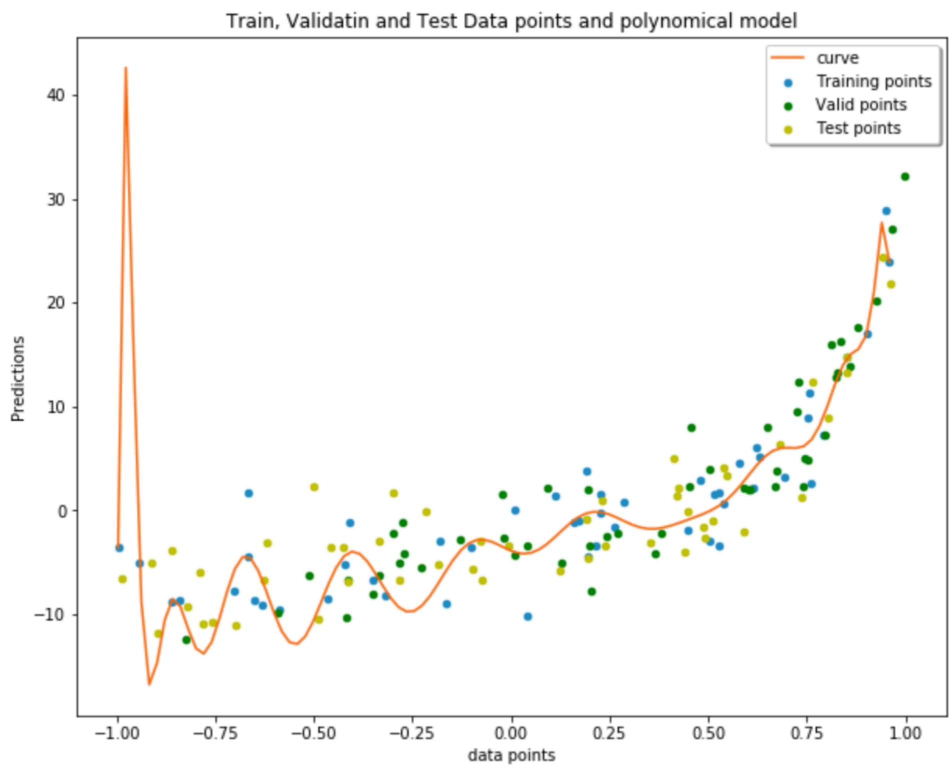

Figure 1- Visualization of the fit and data points

2) In this part, we are supposed to add L2-regularization (ridge regression) to our model. In ridge regression, we add a penalty ($\lambda w^T w$) by way of a tuning parameter (lambda). W and MSE ridge reg is calculated as the following:

$$w_{ridge} = (X^T X + \lambda I)^{-1} X^T y$$

$$\mathrm{MSE(w)} = {}^{1}/_{N} \sum_{\mathrm{i=1}}^{\mathrm{N}} \left(\mathrm{y_i} - \mathrm{x_i^T w}\right)^2 + \lambda w^T w$$

```
def l2_regularization(self, lambda_=0.7):
    self.feat = self.get_poly(self.X)
    I = np.identity(self.feat.shape[1])
    self.w_ridge = np.dot(np.dot(inv((np.dot(self.feat.T, self.feat)+(lambda_*I))), self.feat.T), self.Y)
    print "Parameters W_ridge after L2 regularization corresponding to Lambda {} is: \n {} " .format(lambda_, self.w
    return self.w_ridge;
```

I chose λ between 0 to 1 with step size 0.1, for each λ, calculated training, validation and test MSE. Table 1, shows the comparison between errors. Based on the results, we can see that **λ = 0.1** provide the minimum Validation error that is plausible. So, I calculated test MSE for different λ as well, the minimum error is almost 25.14 that corresponds to **λ = 0.1**. Thus, the best value of **λ** for our problem is 0.1. You can find the variation trends of training and validation errors in figure 2.

Table 1- A comparison between MSEs corresponding to various λ

Lambda	Train MSE	Valid MSE	Test MSE	Description
0	6.47470396778	1417.89812218	50.653704407	Without regularization
0.1	23.5938706187	23.8882630174	25.143325645	Best lambda

0.2	34.6785977962	35.0297033919	36.1220663932
0.3	45.974244803	46.3581789789	47.3569378244
0.4	57.2024072949	57.6089061913	58.5382813516
0.5	68.2823083457	68.7065854341	69.5778511742
0.6	79.1849012965	79.6245642382	80.4438298919
0.7	89.8990785665	90.3529134781	91.1238127955
0.8	100.421217111	100.888646106	101.613474914
0.9	110.751219797	111.232033181	111.912299857
1.0	120.89078928	121.385002321	122.021721272

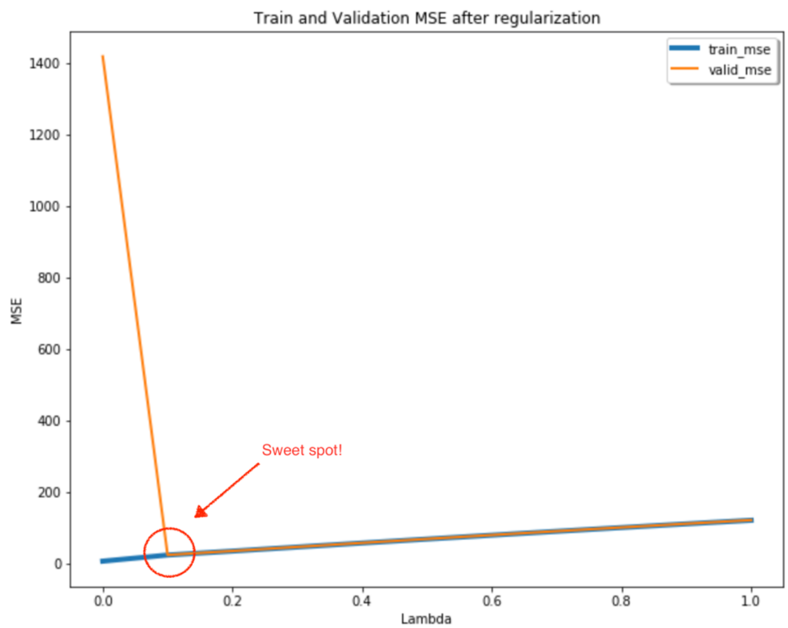

Figure 2- Validation and Train MSE based on λ

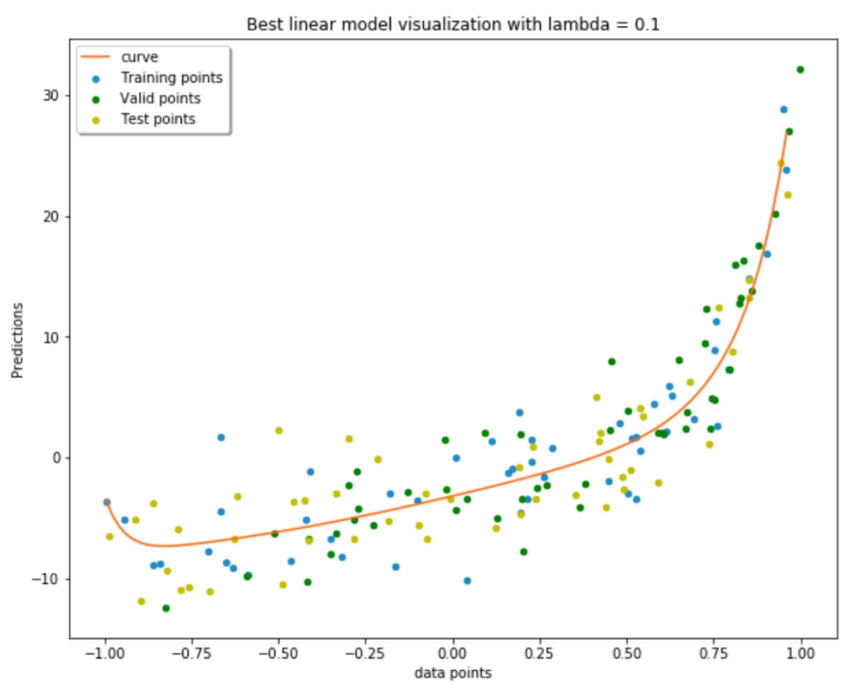

Figure 3- The visualization of the fit of chosen model with λ = 0.1

The fit corresponds to ridge regression with λ = 0.1 is much more generalizable and it hasn't been fitted exactly to the train data points which in some cases can cause outliers. Briefly, this model can predict the overall pattern of the data and has less variance in comparison with higher-degree models.

3) According to the visual information of the figure 3, I am thinking a 2-degree polynomial can be fitted on the data in a way that overfitting doesn't happen and be more generalizable to the test/validation data.

2- Gradient Decent for regression

1) In this part we were supposed to fit a linear regression to the dataset2 by using SGD (stochastic gradient descent). The indicated learning rate (1e-6) was very small and my computer couldn't converge to the local minima with this learning rate, so I considered another alpha which is exp(-6) and showed the MSE on validation set for every epoch with this learning rate:
   I shuffled data before start of training procedure to have a I.I.D (Independently and Identically distributed) dataset. In addition to, I defined epsilon = 1e-10 as a threshold for the parameter (w) convergence.

```
def train_sgd(self,alpha=exp(-6)):
    eps = 1e-10
    self.num_of_epochs = 0
    w = np.ones(2) #Initial Value for W of each iteration on training examples
    last_w = np.zeros(2)
    w_epoch_list = [] # to keep w of 5 diffferent epoch to compare
    w_epoch = w # w of each epoch
    w_last_epoch = last_w
    valid_mse_sgd = []
    train_mse_sgd = []
    while(np.abs((w_epoch - w_last_epoch)[0]) > eps and np.abs((w_epoch - w_last_epoch)[1] > eps)):
        w_last_epoch = w_epoch
        for i in range(self.X.size): # traverse training data points to calculate w for each instance
            self.feat = np.asarray([1, self.X[i]]).reshape((1,2))
            predicted_y = np.dot(self.feat,w)
            loss = predicted_y - self.Y[i]
            gradient = 2*np.dot(self.feat.T,loss)
            w = w - alpha*gradient  #Update W in each iteration
        w_epoch = w #update W of an epoch
        self.num_of_epochs = self.num_of_epochs + 1
```

The validation MSE is: 0.0753326580318
With learning rate: 0.00247875217667
Total number of epochs: 132
Learned W is: [3.56594113 4.30778968] The first one is w0 and and the second is w1.

The learning curve over epochs is showed in figure 4, as the epochs increases the validation and training MSE also increase. This is matched with our intuition from SGD, as we use more data, the model will achieve a deeper understanding of the dataset and w will converge the the stop condition. Also, as we expected the validation error is less than training error. It means, the model works well in predicating new data.

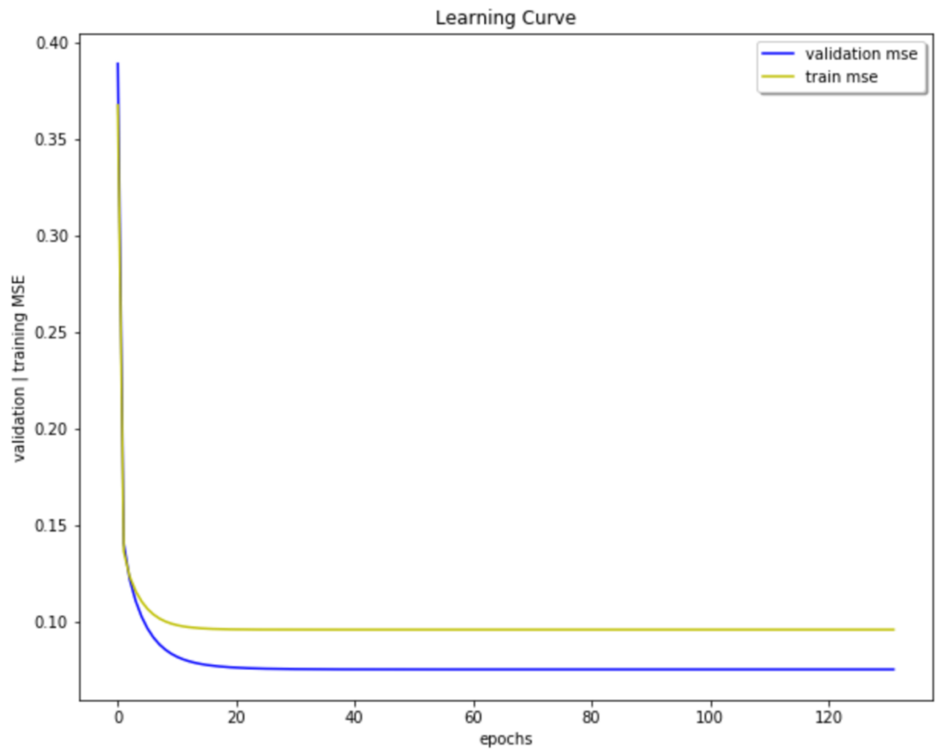

Figure 4- Learning curve over epochs

b) I tried 6 different step sizes and chose the best one based on validation MSE:

Table 2- Validation MSE of various learning rates

Learning rate	Validation MSE	Number of epochs	w0	w1
1	1.03694912146e+171	1	-3.21581135e+85	-5.85458613e+82
0.1	0.0783222829635	5	3.51631257	4.32429332
0.01	0.074456505501	36	3.57484928	4.31417041
0.001	0.0740845113151	313	3.57609302	4.32016513
0.0001	0.0740697062725	2757	3.57650401	4.32029457
0.00001	0.0740702984552	23912	3.57652083	4.32027626

The best learning rate cannot be specified generally since this hyper parameter is very problem dependent and the application of our problem. In this case, according to the above table learning rate = 0.0001 results in the lowest validation MSE among others, so I chose this learning rate to train the best model and showed the test performance as the following:

```
Test MSE of the best linear model is:0.0692334507436 with learning rate = 0.0001
```

Figure 5- A screenshot from Jupyter notebook console

c)
In this part we were supposed to visualize the fit for 5 different epochs during training process:
I divided total number of epochs by 5 and chose epochs evenly to cover all the spectrum of training process and have a better visualization. As we expected, as we go through more epochs the model fit the data points more accurate and with less variance.

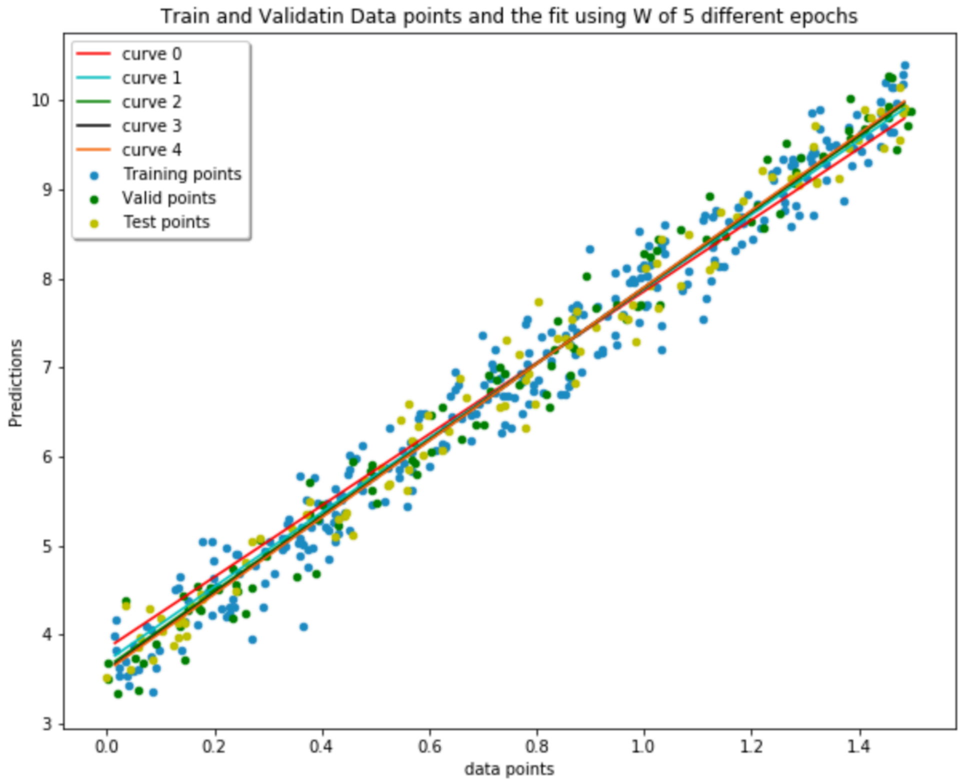

Figure 6- 5 visualization of the fit over epochs

3- Real Life Data Set

1) In this part, we are supposed to fill missing values in the data set. There are different ways for handling missing data, each method has some pros and cons, and there is no best solution for this problem, it really depends on the application and the criteria that we care more about. One of the simplest strategy for handling missing values is to remove the the rows containing missing data. This solution can be very limiting since it will remove all the corresponding instances and the number of examples will decrease. If dataset has a large number of missing values, we may loose many examples.

Another approach can be interpolation of missing values based on their neighbour points, this method has more pros in comparison with discarding methods, at least we won't loose any data. However, this approach needs more computational and time complexity. In the cases that time is more important criteria, we may prefer mean or ignoring approaches.

The mean approach indicated in the question is another way to handle missing values. In my view, using the column mean for filling missing values is a good approach specially when we have a lot of missing values and a high dimension of data. It's an efficient method in terms of time and computational complexity. But this method results in biased parameter estimates. All of the imputation methods underestimate standard errors. Since the imputed observations are themselves estimates, their values have corresponding random error.

In conclusion, choosing a method for handling missing values is a tradeoff between different criteria.

Here, I used the feature mean to handle the missing values in each column, first I traversed trough each column and checked if there is any NaN value, if yes I removed it from that column and saved its index. Then calculate the mean of the column with missing values (NaN) and place the mean in the 'Nan' indices.

```
____________________________feature number:1____________________________
size of feature 1 before delete NaNs:1994
Before Delete NaNs:[ nan  nan  nan ...,  nan  nan   3.]
number of nan for this feature:1174
Size of feature 1 column after delete NaNs:820
After Delete NaNs:[   7.   79.   21.    9.   19.   21.   11.   17.    7.  133.   35.    7.
   79.   35.   15.   49.   17.   53.  141.    3.   61.   21.  750.    3.
    9.   45.   91.    7.   15.   11.   41.    5.   11.    3.   13.   17.
    7.   29.   23.   41.    3.    9.   27.   41.    5.   17.  790.   75.
  113.   17.   35.   17.    1.   25.   45.   39.   23.    9.    3.    5.
   33.   43.    3.   17.    3.  770.   17.   55.   45.   77.    7.   93.
   39.   99.   23.   35.   39.    7.  141.   29.   11.   63.   39.   35.
    5.  165.  139.    1.   27.    7.  153.   79.    3.   17.    5.    1.
  133.   99.   17.   27.   71.   55.   17.   27.   43.   95.    3.    9.
  740.    7.    3.   69.   91.   41.   23.  163.   31.   77.   35.   27.
  510.    3.  133.    7.   13.   27.   71.   27.   61.   87.   23.    1.
  101.    3.   13.   27.   31.   11.   43.   35.   49.   71.   91.    7.
```

Figure 7- Feature 1 specification(with 1174 missing values) this feature has been removed in regression

```
____________________________feature number:21____________________________
size of feature 21 before delete NaNs:1994
Before Delete NaNs:[ 0.56  0.65  0.55 ...,  0.73  0.41  0.21]
number of nan for this feature:0
Size of feature 21 column after delete NaNs:1994
After Delete NaNs:[ 0.56  0.65  0.55 ...,  0.73  0.41  0.21]
```

Figure 8- Feature 21 specification(without any missing value)

```
The number of all no-missing features: 101

_________________Data after replacing all nan elements:__________________

[[  2.70000000e+01   3.00000000e+00   1.27000000e+04 ...,   4.40438871e-01
    0.00000000e+00   1.95078370e-01]
 [  5.60000000e+01   5.88268293e+01   4.61883366e+04 ...,   4.40438871e-01
    0.00000000e+00   1.95078370e-01]
 [  2.40000000e+01   5.10000000e+02   4.00000000e+03 ...,   0.00000000e+00
    3.60000000e-01   3.40000000e-01]
 ...,
 [  4.20000000e+01   4.50000000e+01   8.69680000e+04 ...,   4.40438871e-01
    0.00000000e+00   1.95078370e-01]
 [  5.10000000e+01   7.70000000e+02   6.80000000e+04 ...,   0.00000000e+00
    6.00000000e-01   1.20000000e-01]
 [  3.90000000e+01   5.88268293e+01   4.61883366e+04 ...,   4.40438871e-01
    0.00000000e+00   1.95078370e-01]]

Data shape:(1994, 127)

To check if there is any other Nan(exisiting Nan indices):[]

^^^^^^^^^^^^Congratulation!^^^^^^^^^^^^

-------There is no other NaN element in the dataset!--------
```

Figure 9- A screenshot from Jupyter Notebook to show data after filling

2)

In this part, we should use the filled data in the previous part and fit a linear regression model to data points.
The first 5 features are non- predictive and 4^{th} one is string so the mean of this features is going to be NaN!
I assigned zero for the mean of this feature and filled the feature 4^{th} with 0. I discarded the first 5 features since they cannot help us in prediction, one of them is nominal and most of its values just occur once and the other 4 are non-predictive. So, from now on we have 123 columns (1 goal and 122 features).
I divided data to 5 different files that are going to be used iteratively as the train set and validation set. We have 5 iterations as 5 runs, each of them results in one vector of W with size 122, I printed all of them in Jupyter notebook, but here are just the parameters, training and validation MSE of run 1 and 3:

```
________________________Model Training for run 1_________________

Learned Parameters(W):
[  1.12717518e+00  -1.02958264e-04  -5.46630846e-02   4.70154994e-03
   7.55846378e-03   2.01777718e-02  -1.68425389e-02  -1.32452005e-02
   2.36563365e-04  -6.95331687e-02   2.97263915e-02   5.06120850e-03
   8.86526926e-02  -4.41858849e-03  -1.02025809e-01   1.15618700e-02
   1.38558591e-03   1.10830618e-02  -3.55877026e-02   1.17340414e-02
  -9.63327844e-04   7.14076209e-02   6.05898810e-02  -4.23800746e-02
  -8.53530524e-04   3.26245288e-03   2.87864644e-03   1.72713453e-03
  -5.55123475e-03  -3.45639637e-02  -1.79100775e-02  -1.04966357e-02
   9.93865601e-04  -3.99608864e-02  -6.84619570e-03  -9.95828627e-03
   9.94233409e-03   1.16437621e-02  -3.20810082e-03   1.90988078e-02
  -4.25883429e-02   1.68299772e-03  -9.53526537e-04   2.38241217e-02
   1.92451900e-02  -2.17749995e-02   1.08814754e-03   2.06906232e-02
   7.56476594e-03   3.53836361e-03   1.21983801e-03   1.84951601e-01
   7.46557527e-03  -1.19091824e-01  -9.89695676e-04   2.50039574e-03
  -1.13319182e-02   4.83141128e-03  -3.73750081e-02  -1.49623999e-02
   9.55609829e-02   3.25968648e-03  -1.55112609e-02  -3.02527739e-02
  -6.85452137e-02   7.23001058e-02  -5.47817240e-02   2.30355638e-02
  -1.47279040e-02  -1.02748072e-01   4.62276069e-02  -1.40874261e-02
  -5.50419562e-03  -4.34918977e-02  -5.45606490e-03   9.87883011e-02
   9.53850577e-03   5.66288394e-04   3.55796068e-03   1.58484770e-02
  -5.65162111e-03   7.11506640e-02  -5.56122194e-02  -7.01804041e-04
   1.71688375e-02  -3.05925841e-03  -1.09269057e-02  -2.05774092e-03
   2.72823969e-03  -6.01248348e-04   4.72868735e-03  -5.50771088e-02
   1.01036748e-02  -1.06135508e-02  -4.76573654e-03  -6.57817054e-03
  -5.30383352e-03   9.35247149e-03  -2.16056970e+00   1.49725348e+00
  -9.32621595e-01  -4.21504570e-02  -6.99087172e-02   4.64325606e-02
   2.32223738e-02  -5.36841352e-01   3.01949049e-02  -2.02620110e-01
   1.60939624e-02   1.25624796e-01   7.65108889e-02  -2.99434431e-01
  -1.47872645e-01  -1.26877973e-02   1.54880026e-02   9.99260503e-03
   1.82389243e-03   8.87106653e-05   7.00167792e-02   1.35491320e+00
  -1.64506538e-02   1.27204107e-02]

Train MSE:0.000499460468776, Valid MSE:0.000578263906463
```

```
________________________Model Training for run 3___________________

Learned Parameters(W):
[  1.15124833e+00   1.18265081e-04   5.88833341e-02  -4.73627351e-03
   1.03007020e-02   1.88333320e-02  -1.70194958e-02  -4.70795108e-03
  -5.13161610e-04  -8.20899552e-02   3.58329766e-02  -4.42623256e-03
   9.47615855e-03  -3.01459552e-03  -9.76124269e-02  -6.88616834e-03
   1.94627427e-03   2.86748960e-03  -3.70220589e-02   5.80719375e-03
   2.79801203e-03   8.97236485e-02  -1.83815326e-02   1.16862135e-02
   9.21502020e-04   8.55999324e-05   2.55946398e-03   6.45508611e-04
  -1.65398202e-03  -4.33115822e-02  -2.17682139e-02   2.26861225e-03
  -1.14772586e-02  -3.79568394e-02  -9.82420135e-03  -5.35059269e-03
   9.71834200e-03   6.34218421e-03  -1.82472804e-04   1.40232884e-02
  -6.47851188e-02   1.01657335e-02  -3.15277821e-02   7.21517184e-02
  -8.41644262e-03  -5.72372346e-02   4.62452749e-02   1.58800982e-02
   4.95019816e-03   5.21133563e-03  -6.52542837e-03   1.75998572e-01
   2.31902518e-03  -1.41085100e-01  -4.53483245e-03   8.01050615e-03
  -1.46180840e-02   7.31446974e-03  -1.27770749e-02  -3.71448656e-02
   1.19473822e-01  -3.49900702e-02  -3.41922178e-03  -2.98457583e-02
  -2.64570581e-02   2.62928393e-02  -5.34210576e-02   6.65092813e-02
  -1.73303522e-02  -1.67004270e-01   4.38301833e-02  -8.15958884e-03
  -6.95670998e-03  -3.17898127e-02  -1.22402924e-03   1.66823559e-01
   1.00930906e-02  -3.75758646e-04   1.54567329e-03   1.45859051e-02
  -1.57780276e-03   4.84668104e-02  -9.03935239e-02   4.32529702e-02
   1.91872293e-02   1.37413635e-03   3.04557643e-03  -1.31357465e-02
   8.07533855e-04   1.16422386e-03   1.36185676e-03  -5.76119553e-02
  -6.18580080e-04   3.54155804e-03  -8.31263535e-03  -4.36866873e-03
  -3.50201248e-03   8.72341255e-03  -2.21425186e+00   1.82448775e+00
  -9.67942075e-01   1.85753309e-01  -1.22938139e-01   5.99541107e-02
   5.78342345e-03  -1.07876878e+00   5.10714350e-02  -1.92832929e-01
   4.44886671e-02   1.63514434e-01   8.41234908e-02  -3.05966226e-01
  -5.74631386e-02  -1.23677823e-02   2.88207214e-02  -1.30685051e-02
  -2.73930676e-03   3.36544064e-04   4.74092504e-02   1.37162823e+00
  -4.19836955e-02   1.19882347e-03]

Train MSE:0.000491000037909, Valid MSE:0.000576816548312
```

The average test MSE achieved over 5 runs is:

```
_______The Average of Training MSE of 5 folds cross validation is: 0.000457645735173

_______The Average of Validation MSE of 5 folds cross validation is: 0.000830830088048
```

In each run I have a different look of learning curve because I shuffled data in each run, so each time the outlier points are in a different set!

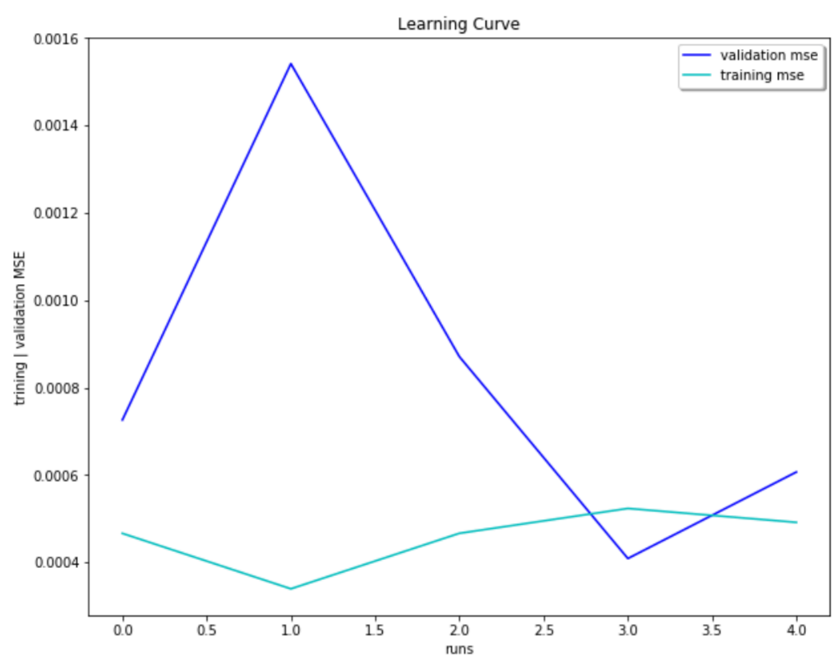

Figure 10- Learning curve over 5 runs

3)
In this part we are going to use ridge regression on the data, so I defined a list of lambda to train different models. There are 10 different values of lambda, in each run I trained a model by using these 10 lambdas, we have 5 runs, so at the end I have 50 models with 50 different learnt parameters. Here, I provided w of two different lambdas of different runs. But all of 50 parameters have been printed on Jupyter Notebook as well.

```
________________________Model Training for run 1 with lambda:0.1_________


Parameters W_ridge after L2 regularization corresponding to Lambda 0.1 is:
 [ -1.43195263e-01  -1.87346365e-04  -9.16858092e-03  -2.60098382e-03
   6.68542130e-03   1.19752162e-02  -9.95371208e-03  -1.53964670e-02
  -5.99609659e-04  -3.33304200e-02   1.65068216e-02  -2.44988885e-02
   2.34754348e-02  -4.88612664e-03  -3.45500787e-02  -2.45483371e-03
  -1.01823682e-03  -6.01378825e-03   7.16378519e-03  -1.16836680e-03
  -4.47265294e-03   5.38752378e-02  -8.86020048e-03   9.15683789e-03
   2.23539137e-03   2.25763522e-03   1.48014831e-04  -1.06356753e-03
  -3.12803876e-03  -6.02916166e-02   1.12074989e-03   3.98927585e-04
  -1.62045346e-02  -3.53839888e-02  -9.45139726e-03  -1.14062220e-02
   1.90733774e-03   4.24707567e-03   1.07952555e-02   1.96669366e-02
  -4.83969390e-02  -1.49238246e-03  -4.19487115e-03   4.08628184e-02
   1.54923459e-02  -2.40407252e-02   2.80725055e-02   1.35708437e-02
   4.14527971e-03   5.42017812e-03  -6.34301835e-03   1.03083704e-01
   2.46826789e-03  -5.15220139e-02  -4.24834898e-03   2.57272304e-03
  -9.81944977e-04  -4.57223442e-03  -1.72294083e-02   1.44582166e-02
  -1.96833488e-02   6.73799818e-02  -1.51452498e-02  -2.46380027e-02
   3.47669846e-03  -3.08013768e-03  -3.07473228e-02  -4.34541679e-03
   9.03815812e-04  -5.04156240e-03   3.45019625e-02  -4.22291365e-03
  -4.29775629e-03  -8.10216671e-03   4.17236162e-03  -2.76017482e-03
  -4.35676031e-04   3.55859004e-04   5.34382765e-03   1.03477033e-03
  -1.80275562e-03   2.47302381e-02  -3.03963107e-02   5.59911975e-03
   1.48774998e-02  -7.06674986e-03  -1.90072975e-02   1.07743429e-02
   6.12798660e-03   5.21911531e-03  -6.28006629e-04   1.15151689e-03
   2.13973710e-02  -1.81342500e-02  -1.23764955e-02   1.72224856e-03
   1.24735602e-03   1.06503749e-02  -3.25287068e-01   3.53058125e-01
   1.83936381e-01   1.83637239e-01  -1.66214134e-01   9.54178162e-02
   2.20532482e-02   3.54032742e-01   4.18310470e-02  -8.86681898e-02
  -1.06090404e-02   6.47350284e-02   1.04549656e-01  -1.23368549e-01
  -1.99059168e-01  -3.93523045e-05   3.26134496e-02   1.30081756e-03
  -2.42992340e-03  -3.53723065e-03   5.40284737e-02   7.46754124e-01
  -3.10423360e-03   1.01974664e-02]
After Regularization with lambda 0.1:
 Train MSE:0.117126353522, Valid MSE:0.118038337611
```

```
_________________________Model Training for run 3 with lambda:0.6__________


Parameters W_ridge after L2 regularization corresponding to Lambda 0.6 is:
 [  5.67307113e-03  -9.24316225e-06  -3.60915983e-03  -8.39262879e-03
   1.03696153e-02   1.82644099e-02  -1.27625055e-02  -1.00242075e-02
   1.05974999e-02  -3.06179221e-02  -9.41540446e-03  -1.40439908e-02
   1.41548948e-02  -4.81705990e-03  -3.21341056e-02  -6.83305566e-04
   1.04960870e-03   3.75547812e-03  -1.64482082e-02   1.47324705e-02
  -2.04294610e-04   3.94241881e-02   1.12279013e-02   4.24375176e-03
  -4.39668383e-04   4.54559109e-03   4.02454702e-03   2.25596841e-03
  -1.74106123e-03   8.98350429e-03  -2.21304910e-02  -5.14048581e-03
  -3.69812781e-03  -3.12936149e-02  -1.04169590e-02  -1.13705265e-02
   8.50485026e-03   1.42288884e-02  -3.70633191e-03   2.16520698e-03
  -3.04914066e-02   1.44884937e-02   1.06940409e-02   3.89113946e-03
  -5.63080368e-03  -2.68689243e-02   2.51152203e-02   2.67432557e-02
   6.70649779e-03   4.30406028e-04  -1.08211853e-03   2.72308966e-02
   7.19279562e-03   1.14099478e-02  -7.90627682e-03   6.13012765e-03
  -1.76445234e-03   1.86775069e-04  -2.04302835e-02  -2.69912489e-03
   2.53597206e-02   2.53119107e-02  -2.12272623e-03  -2.87969694e-02
   5.33169737e-03  -1.65494512e-03  -1.36836148e-02  -8.01078348e-03
   6.03841303e-03  -1.24234849e-02   3.60240690e-02  -2.93905158e-03
   4.28149328e-05  -1.21265418e-02   8.16570899e-04  -1.55400438e-04
   9.05515735e-03   6.29127241e-04   2.87298283e-03   1.50374144e-02
  -3.66191529e-03   2.20735609e-02  -7.07949417e-03  -1.21795925e-02
   1.38372472e-02   5.05042271e-03  -2.45058670e-02  -5.63081176e-03
   1.41649843e-02  -2.09935999e-03   5.20285717e-03  -2.67766362e-02
   2.31937313e-02   5.14408247e-03  -1.18067087e-02  -7.96128071e-03
   1.99822618e-03   1.17828353e-02  -1.08416341e-01   3.09397038e-01
   1.15112132e-01   2.53299921e-01  -9.57754658e-03   9.59338702e-02
  -3.82418716e-02   3.09413984e-01   9.89726749e-03  -1.24312452e-01
  -6.58350473e-02   1.65784221e-02   1.01547061e-01  -1.28843264e-01
  -1.55892918e-01   6.24456003e-03   5.49315222e-02  -1.61138723e-02
  -6.83413171e-03   7.01945661e-04   5.26670047e-02   2.54076657e-01
  -3.95389320e-02   1.50411608e-02]
After Regularization with lambda 0.6:
 Train MSE:0.274031588829, Valid MSE:0.274095382915
```

According to table 3, in ridge regression as lambda decreases, the mean squared error decreases. It seems,

Table 3- Average valid MSE and average train MSE after regularization with different lambdas

Q	Avg validation error over 5 runs	Avg training error over 5 runs
1	0.19269546238	0.192422576422
0.9	0.180209529243	0.179932449039
0.8	0.167233589518	0.166951765901
0.7	0.153657065996	0.153369789188
0.6	0.139317814719	0.139024135958
0.5	0.12396816459	0.123666762216
0.4	0.107211146736	0.106900082382
0.3	0.0883730383618	0.0880492769431
0.2	0.066232029770	0.0658903989025
0.1	0.0384319861076	0.0380626621801
0	0.000872209162432	0.000454690993524

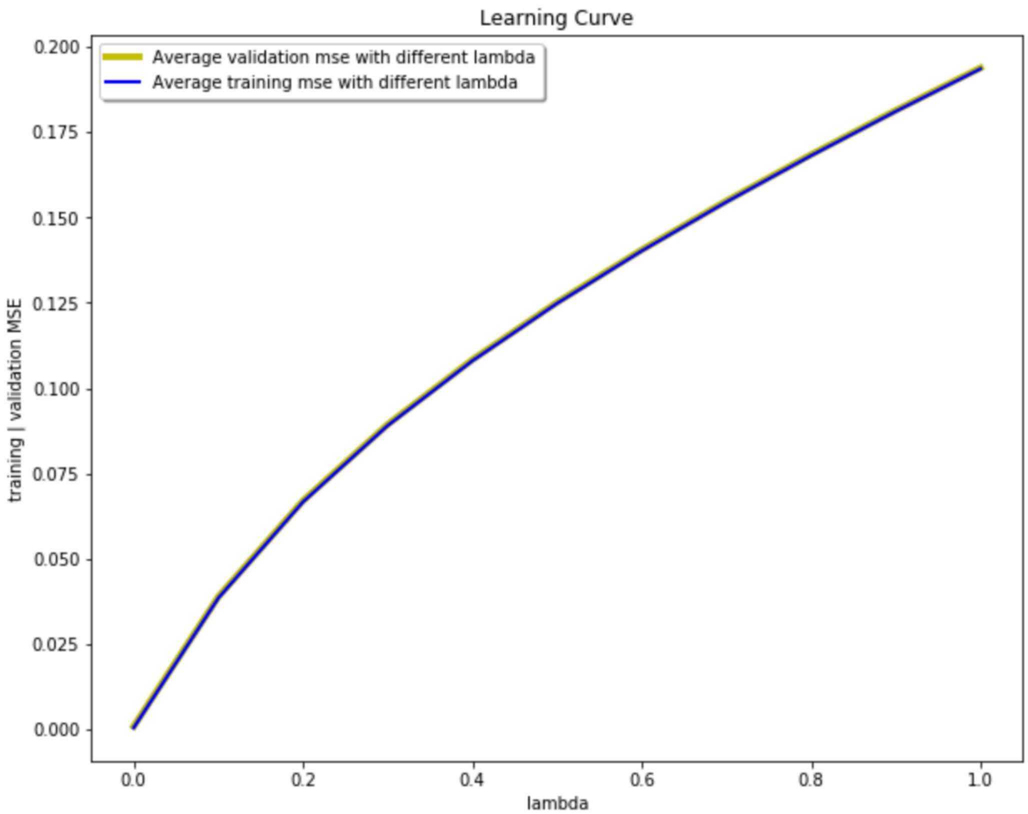

Figure 11- Learning curve after ridge regression

Ridge regression reduce the model complexity more than reducing less relevant features. As lambda gets larger, the bias is unchanged but the variance drops. So, the complexity of model decreases.
However, the drawback of ridge is that it doesn't select variables. It includes all of the variables in the final model. In addition to, it drives few features to zero, so we cannot use the ridge model for feature selection but Lasso effectively sets the weights of less relevant input features exactly to zero. So, it is a tradeoff between reducing model complexity and reducing less relevant features.