

Assignment Report

This is the report of assignment 2 which will be tackling the explanation of implementing the assignment in various techniques of machine learning wide scope. The assignment description directed us to use the techniques discussed in lectures 5 and 6 thoroughly. Given the wide range of techniques explained in both lectures, I tried to come up with the most challenging and explorative ones to be implemented in the assignment. However, this report will indulge a detailed discussion on the un-used techniques brought up in the lectures that were not tackled in the assignment in order to state the pros and cons of each and to clarify that these techniques were very well comprehended, but due to time difficulties and following similar techniques that showed similar results they were not purely implemented in the assignment.

Allow me to start first by the problem formulation. The problem that we may face when splitting our data into train and test datasets is the occurrence of overfitting or under fitting; overfitting is when the data set is very highly trained with all the given data points so that any new data point faced will result in an high degree of error and inaccuracy, while under fitting is simply the opposite, where the data used is not very well trained to result in an accurate measurement when being opposed with a new data point unplaced in the training data set. In order to overcome and conquer this problem we may use model selection techniques. The question is what are the different model selection methods? Answering this potential question, there are cross validation, random sampling, polynomial regression and other type of likewise model selection techniques.

Let us kick it off by discussing the first used technique in the assignment, which is the cross validation method. During the course of the implementation done in this assignment, I started with importing and reading the dataset given, which is the house prices training dataset using pandas and to re-assure the validity of this step I plotted the price on the y-axis alongside with living room size in feet measurements present on the x-axis. This is just a validation step, as the models itself will be built using the bedrooms and bathrooms columns data with the price column data. Afterwards, the step of splitting the dataset to train, test and validation data occurred by the aid of `to_numpy()` function to have “numpy” vectors not pandas data frame in order to facilitate the upcoming steps. As proposed in the lecture data were sub-sectioned as follows:

1. 60% training data
2. 20% validating data
3. 20% testing data

This is done mainly for one reason and one reason only, which is to optimize the parameters in theta using the training set for each polynomial degree J of theta. Then finding the polynomial degree d with the least error using the cross validation data and afterwards finally, estimate the generalization error using the test set: $J_{test} \theta = \text{summation}_{i=1}^{m_{test}} (h_{\theta}(x_{test}(i)) - y_{test}(i))^2$. One extremely important remark that was accomplished in this assignment is the feature normalization of the data set to make a good fit when being introduced to our model. In other words data must be normalized to fit a descent and accepted range of data values to result in meaningful results for testing our model. Moreover, bedrooms and bathrooms data have occupied the X values for each train, validate and test data arrays. On the other hand, meaning the Y values, price data values occupied this region for also each of our three sub-sectioned data numpy arrays. Before implementing our model, we shall not miss out on concatenating ones into each of these arrays as a highly vital step that helps facilitating our arrays to be injected into the model; that is a neat way for representing the data arrays.

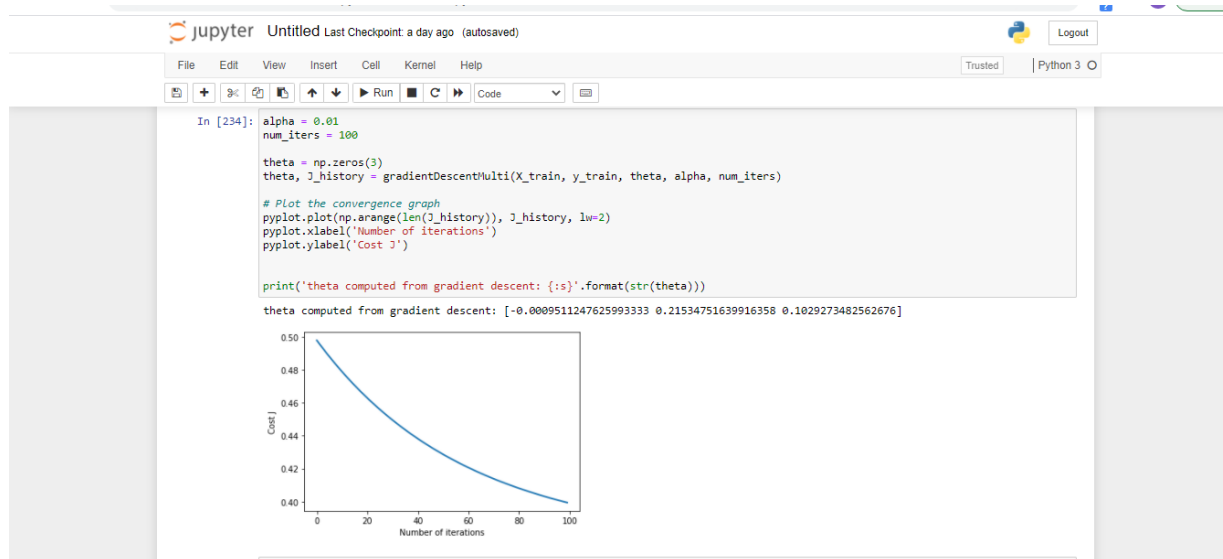
Now let's dive right in to the cost and gradient functions and calls. As per the lectures slides, there was a question that was raised on top of the surface of the table for discussion, which is: **Which polynomial to choose for the hypothesis function?** The proposed and the implemented solution in my assignment was to iterate over various H of theta, also known as the hypothesis polynomials, in order to find a pattern or a minimal error provided from one of the polynomials. One point in favor of this technique is although it may seem repetitive and time consuming, one may easily find a pattern and enhancements when increasing the polynomial of the hypothesis function by a very observant reduction in the error calculated iteratively. This is the exact argument that my implementation stood up for; in other words, other than going through for instance 18 iterations I thought of doing up to 5 iterations finding a pattern and proving a point when calculating the error, which is the clear enhancements proved from the provided models when going up with the polynomial degrees.

However, it was not that simple, as varying the alpha will result in misguided results. Therefore, I followed an approach to keep the alpha stable for example using a value of 0.01 through-out the multiple variations of the polynomial degrees. This will not be it, as afterwards changing the alpha and observing the effect will take place in order to proceed with the validation and testing data with the least possible error of J of theta that we may

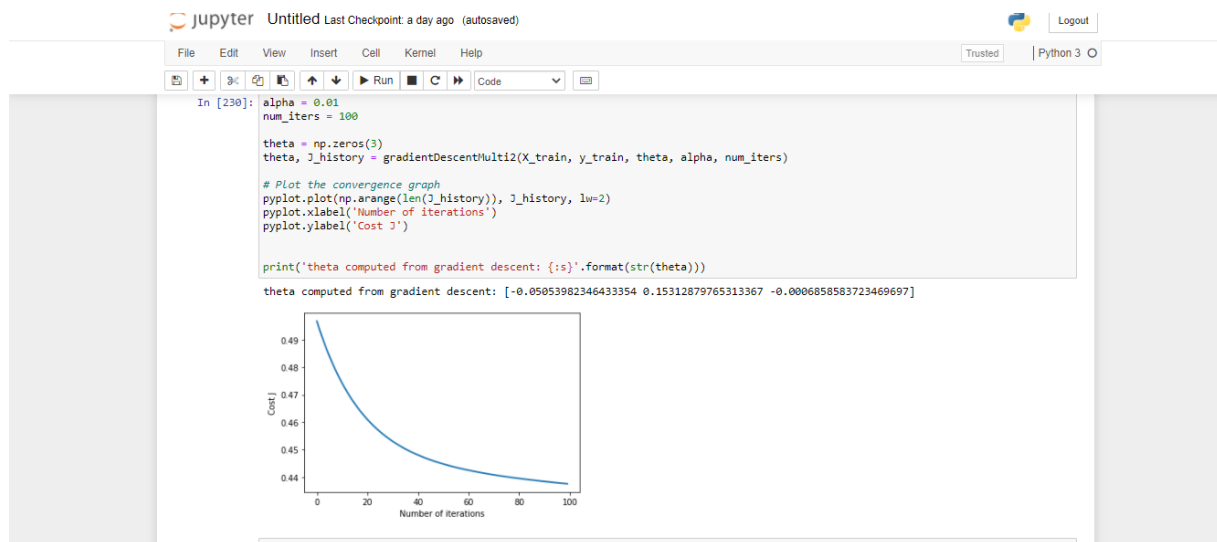
obtain. SO, we need to minimize this cost function resembled in the following equation $J(\theta) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(x^{(i)}) - y^{(i)})^2$. where the hypothesis $h_{\theta}(x)$ is given by the linear model

$$h_{\theta}(x) = \theta_0 + \theta_1 x$$

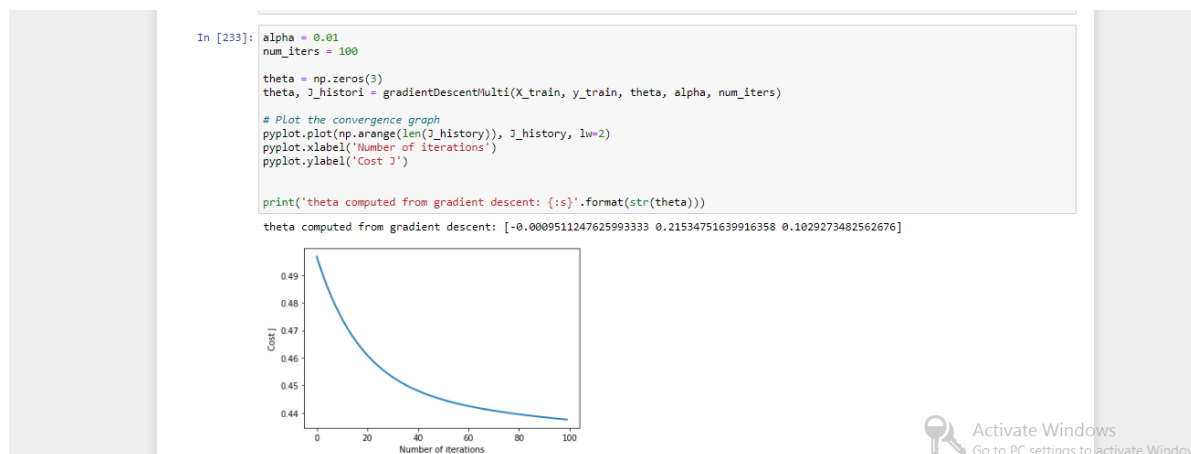
for the first cost function method implemented then calling the function with an alpha of 0.01. The result is being provisioned in the figure below:



The figure shows that for this function the cost or the error is still not optimized and we shall not proceed with these values; if you take a look on the graph, the cost J of θ is approximately reaching 0.5. Further modifications were introduced by increasing the polynomials as previously discussed being a second degree polynomial function and performing the same steps as for a first degree polynomial function with the same value of alpha being in use, and this resulted in a better graph and enhanced/less error values as shown below:



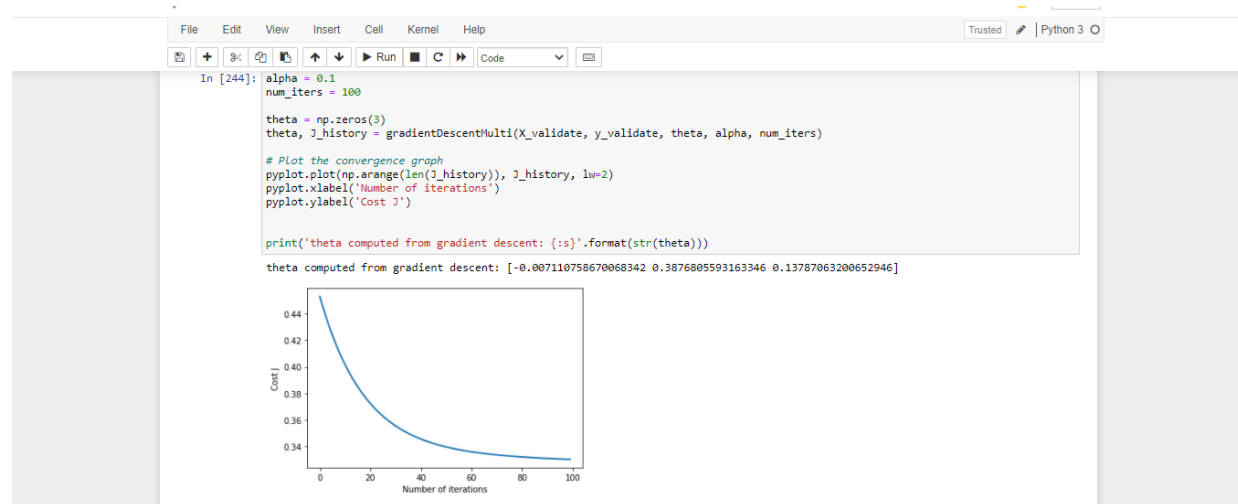
The cost function J of θ was enhanced as you may figure out, even indicated from the graph that did not even reach to a 0.5 value, and also being more smoothed towards the zero. Of course θ computed from gradient descent is clarified also on top of the graph being enhanced as well, so we need not go through the numbers again as it is clearly stated here. Now, let us have a look on the figure appearing below this text:



This figure shows the graph and error values appeared after the 3rd degree polynomial function took place and here we need to hold on for a moment to discuss these results. As per the graph available in the figure, enhancements have been shown, however, enhancements are not that obvious when being compared to change of effect when comparing figure 2 of the second degree polynomials with the first one. If we may conclude anything, it will be that results YES are getting better, but the rate of change

towards a perfect error rate converging to zero may take some few other iterations using the same alpha value provided on all functions. Accordingly, as stated before in this report, we need not iterate tens of times in order to validate our hypothesis or our point, as long as we see a significant enhancements over the course of the iteration processes, we can stand up to this baking our point that we may see a pattern of improvements here.

Moving on to what I call the next phase, which is variation of the alpha values, I have indeed tried changing the alpha and calling the past 3 implemented cost functions as well as the gradient descent again after changing the value of the alpha, and this has effectively resulted in changing the J of thetas and the errors rate of change. In order to illustrate more, given an alpha value of 0.1 this time it has resulted in the following graph and gradient descent theta values:



This was when injecting this alpha value into a first degree polynomial function, however when trying it on a second degree polynomial function this was the outcome:

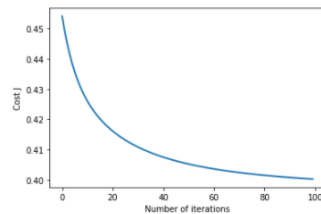
```
In [245]: alpha = 0.1
num_iters = 100

theta = np.zeros(3)
theta, J_history = gradientDescentMulti2(X_validate, y_validate, theta, alpha, num_iters)

# Plot the convergence graph
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J')

print('theta computed from gradient descent: {}'.format(str(theta)))
```

theta computed from gradient descent: [-0.15086500160736255 0.20411016170044877 -0.01853524356216298]



Activate Windows

Given the values and the smoother and lower error values we can conclude that changing the alpha value as well as the gradient descent along with the cost function are both consistently vital for improving performance and decreasing the error here.

For the sake of being diverse, you can refer back to the assignment results, where I tried the same steps on a third and fourth degree polynomial hypothesis function and the previous results were strongly backed up and validated as well. The following figure can illustrate best my words for clarification without the need of going through the graph and the error values results again and again.

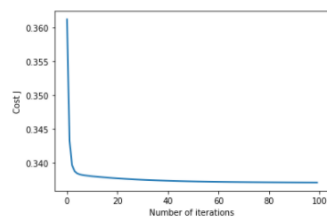
```
In [246]: alpha = 0.1
num_iters = 100

theta = np.zeros(3)
theta, J_history = gradientDescentMulti3(X_validate, y_validate, theta, alpha, num_iters)

# Plot the convergence graph
pyplot.plot(np.arange(len(J_history)), J_history, lw=2)
pyplot.xlabel('Number of iterations')
pyplot.ylabel('Cost J')

print('theta computed from gradient descent: {}'.format(str(theta)))
```

theta computed from gradient descent: [-0.04569470983292006 0.0921567983101148 0.01738143961581422]

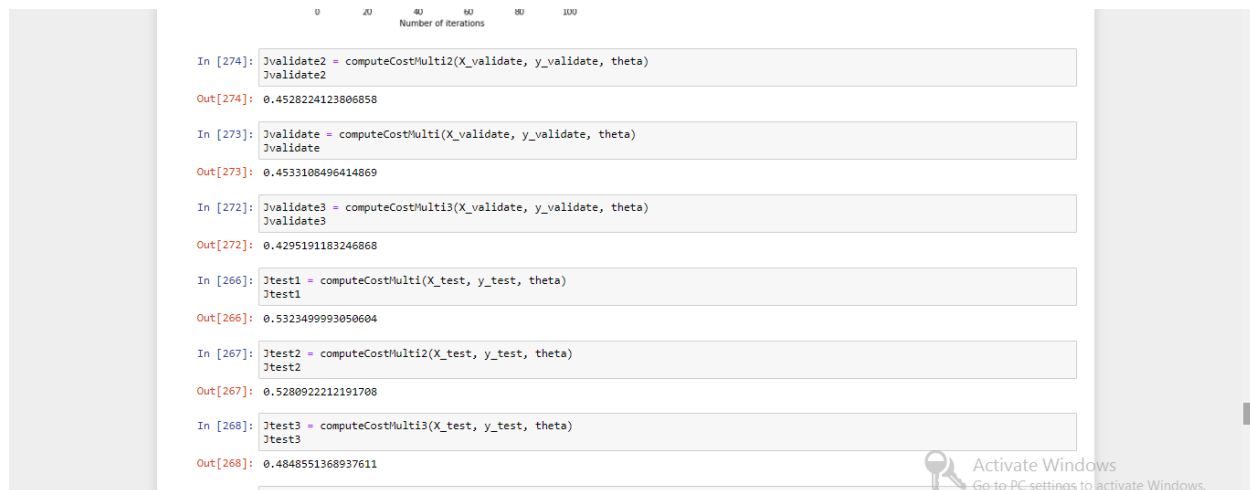


```
In [249]: alpha = 0.01
num_iters = 100

theta = np.zeros(3)
```

Activate Windows
Go to PC settings to activate Windows.

So what is the following step? I moved on to validate the results with the cross validation sub-sectioned data; being also iterated to find out the measurements of the error and J of theta values. Afterwards, to avoid being too detailed oriented as you may refer back to the assignment results, the test data has come up front to take its place as well in the iteration processes validating that the increase of polynomial degrees do enhance the resulted error and the cost function of course.

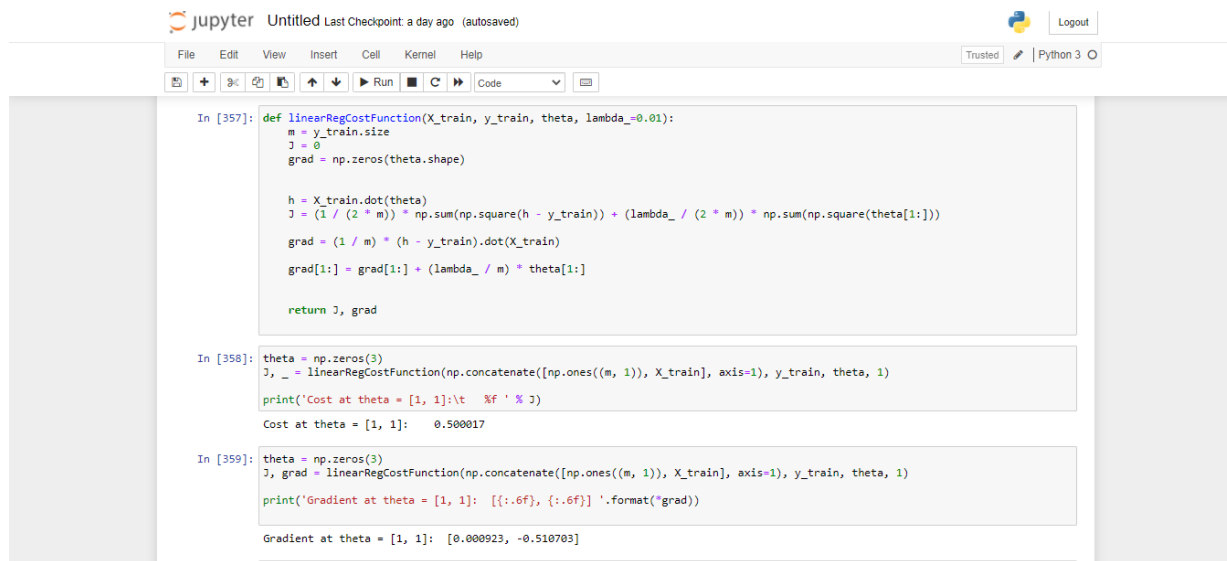


The figure above this text clearly indicate the results in numbers leaving no a shadow of doubt to the experiment done.

To be clear, random sampling also might have taken place here in more depth being named as k-fold sampling indicating a value of 10 for the K present. However, it was not implemented, to be frank due to 2 main reasons, the first is that the previous method is far more accurate [1] and has multiple feature handling and the other reason can be clarified explicitly as time-wise.

Now allow me to move on to the regularization techniques, but first let us clarify what is regularization in machine learning? In mathematics, statistics, finance, computer science, particularly in machine learning and inverse problems, regularization is the process of adding information in order to solve an ill-posed problem or to prevent overfitting. Regularization can be applied to objective functions in ill-posed optimization problems. The added information here or let me say variable is the lamda.

Lamda can be equal to zero or simply the main struggle is to find the perfect lamda value.



```

In [357]: def linearRegCostFunction(X_train, y_train, theta, lambda_=0.01):
            m = y_train.size
            J = 0
            grad = np.zeros(theta.shape)

            h = X_train.dot(theta)
            J = (1 / (2 * m)) * np.sum(np.square(h - y_train)) + (lambda_ / (2 * m)) * np.sum(np.square(theta[1:]))

            grad = (1 / m) * (h - y_train).dot(X_train)
            grad[1:] = grad[1:] + (lambda_ / m) * theta[1:]

            return J, grad

In [358]: theta = np.zeros(3)
J, _ = linearRegCostFunction(np.concatenate([np.ones((m, 1)), X_train], axis=1), y_train, theta, 1)
print('Cost at theta = [1, 1]:\t %f %f' % J)
Cost at theta = [1, 1]: 0.500017

In [359]: theta = np.zeros(3)
J, grad = linearRegCostFunction(np.concatenate([np.ones((m, 1)), X_train], axis=1), y_train, theta, 1)
print('Gradient at theta = [1, 1]: [{:.6f}, {:.6f}]'.format(*grad))
Gradient at theta = [1, 1]: [0.000923, -0.510703]

```

This figure and the cell blocks above in the ipynb file are the perfect illustrations when lamda is equal to zero and when it is equal 0.1. Through the J function indicated above and placing lamda at a value equals to 0.1 the max error that we may admit can be a value of 0.5 and also the range is stated below.

Last but not least, you can find the lines of code that perform a learning curve illustration, however, I had an error that I could not resolve so unfortunately I could not come up with the actual graph of the learning curves to illustrate the bias vs variance topic or method using a vectors of lamda not trying it out sequentially as illustrated in the previous paragraph.

References:

[1]https://www.researchgate.net/publication/319170495_Dependency_Analysis_of_Accuracy_Estimates_in_k-Fold_Cross_Validation