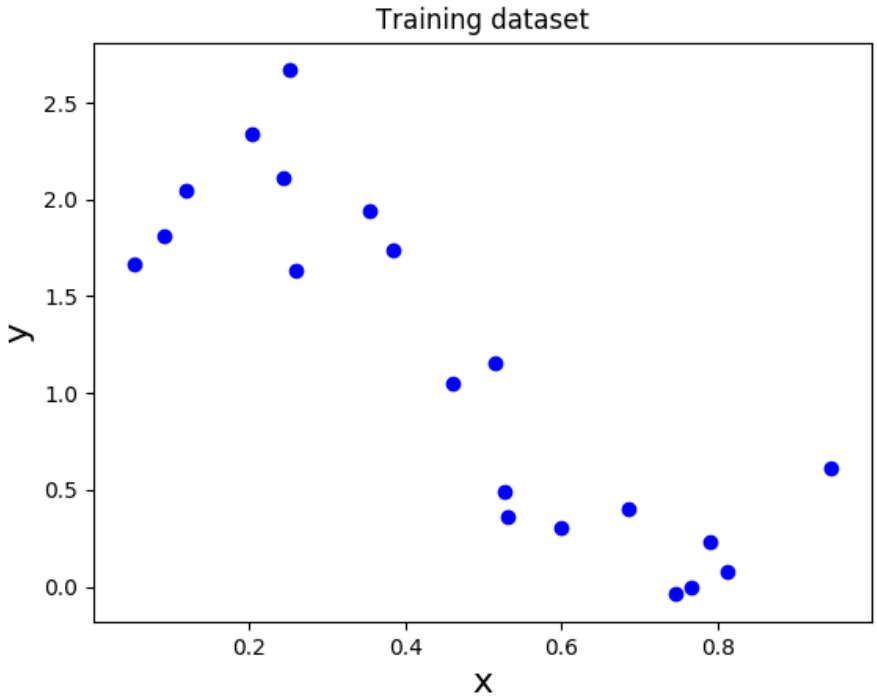
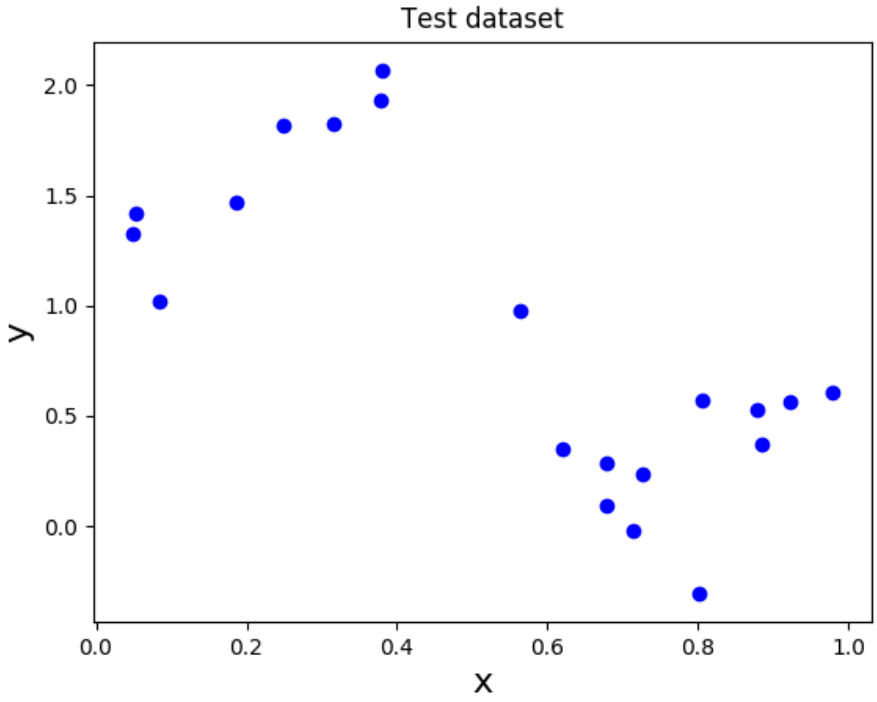
Problem 4

(a) Solution: After visualizing the data using 'matplotlib.pyplot', we get the following charts:





Clearly, the two plots have well-separated positive and negative datapoints. While there are two data points at y = 1.0 in the training dataset and the test dataset (at the margin of the proposed separation line), they are still > 1. Since there is a separation in the positive and negative points, linear regression will be successful in predicting the data (i.e. training error and test error will be low).

(b) Solution: The new X matrix was created by concatenating a column of ones to the front of the original matrix, in *regression.py*

(c) Solution: The equation $y = wX + B$ was used to perform matrix multiplication and predict class values.

(d) Solution: First, the two functions *cost()* and *fit_GD()* were completed using the problem specifications. The algorithm was run with different learning rates, and the following table was outputted:

Step size	Coefficients	Number of iterations for convergence	Final cost	Time (s)
0.0001	W_0 = 2.27044798 W_1 = -2.46064834	No convergence	4.0863970	0.32016
0.001	W_0 = 2.44640680 W_1 = -2.81635300	7021	3.9125764...94	0.18199
0.01	W_0 = 2.44640703 W_1 = -2.81635347	765	3.9125764...49	0.02208
0.0407	W_0 = -9.4e+18 W_1 = -4.7e+18	No convergence	2.7e+39	0.25675

Periods (...) added to denote differing decimal points between values.

The coefficients seem to be converging to [2.44640703, -2.81635347] i.e. the result with step size = 0.01. As the step size increases from 0.0001 to 0.01, the zeroth coefficient increases gradually and the first coefficient decreases gradually until the converging value. For step size = 0.0407, the step size has exceeded the optimum value and makes the algorithm jump over the minima, thereby resulting in erroneous coefficients.

The total time taken for the algorithm coincides with the number of iterations taken for convergence. (We can look at either one for analysis). Clearly, the optimum step size = 0.01 converges fastest with 0.02208 seconds. With 0.001, the step size is still lower than optimum. Therefore, even though the algorithm converges, the number of iterations and time taken is greater. Lastly, the step sizes of 0.0001 and 0.0407 are too small and too big respectively. Therefore, the algorithm doesn’t converge (all 10000 iterations are used) and the time taken is higher.

(e) Solution: After implementing the function *fit()*, the following coefficients and information were extracted:

Coefficients	Final cost	Time (s)
W_0 = 2.44640709 W_1 = -2.81635359	3.9125764...64	0.00039

The coefficients from the closed-form solution were very close to the coefficients with step size = 0.01. Since the value of the zeroth coefficient has increased from SGD, and the value of the first coefficient has decreased from SGD, it seems that the closed-form coefficients are the converging value. Therefore, this approach has produced better coefficients. This is supported by the cost, which is lower than all experiments with SGD.

The algorithm is faster than SGD as well, finishing in 0.00039 seconds (compared to 0.02208 seconds from SGD). This is because the closed-form solution is just a matrix multiplication, with no need for iterations. However, while this approach proved better for the given dataset, the SGD approach would fare better with a larger, more complex dataset.

(f) Solution: For the proposed learning rate = 1 / (1 + k), the algorithm produces the same coefficients as the closed-form solution (withing a small rounding error). The algorithm, however, still takes 1719 iterations and 0.0505 seconds to converge to the coefficients. Overall, SGD with a fixed step size of 0.01 performs better than a variable step size of 1 / (1 + k), as it converges within fewer iterations. (Some of the extra time consumed can be because of the recalculation of step size in every iteration but this is minimal)

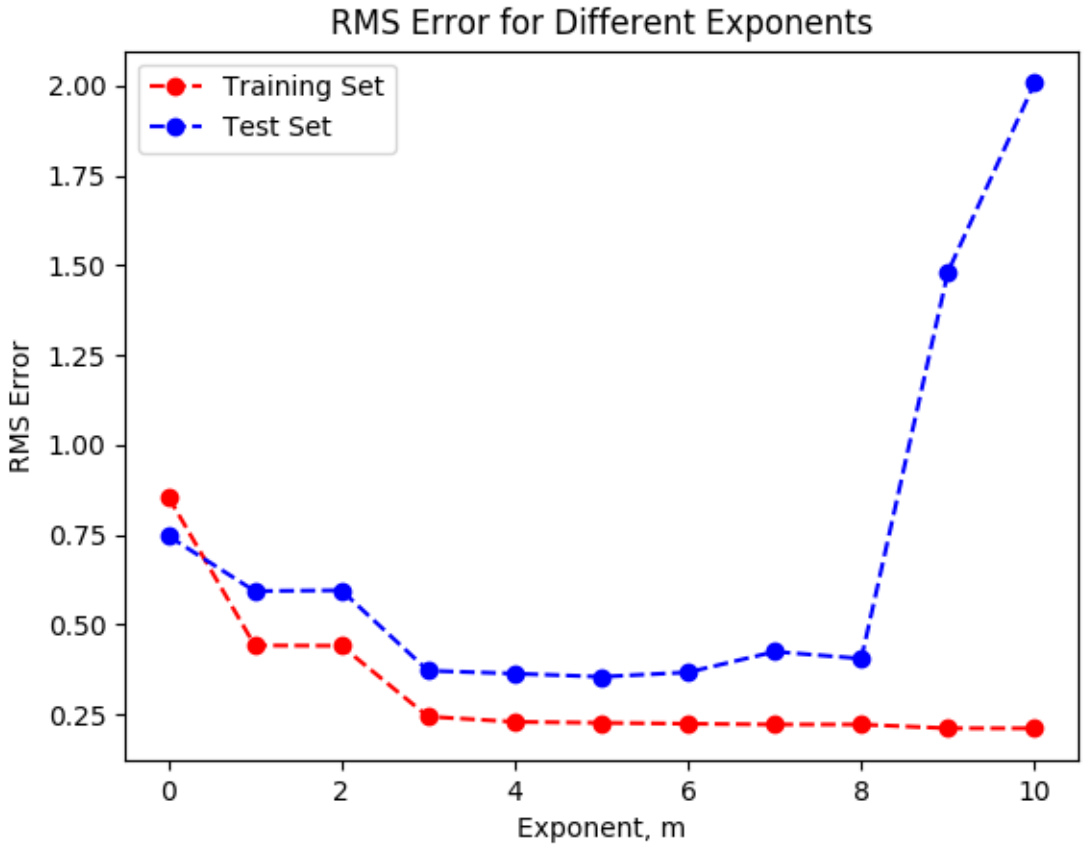
(g) Solution: The function *generate_polynomial_features()* was updated to calculate the polynomial regression matrix X. By looping through each element in X, the increasing exponent is added to the matrix creating the final form as:

$$X = [1\ X\ X^2\ X^3\ \ldots\ X^m]$$

(h) Solution: The cost function J() measures only magnitudes of incorrectly predicted classes. This doesn't give us a picture of overfitting, which can be seen through correct predictions as well.

The RMS normalizes the error such that the standard deviations between actual values and predicted values can observed. This would be a better measure to detect overfitting, as the test RMS and the training RMS can be compared directly. (Overfitting would imply a small error in training data and a large error in test data).

(i) Solution: After training and fitting the model with different polynomials, the following graph of RMS error was produced:



In my opinion, the polynomial regression model with $m = 5$ has the best fit with the dataset. Both the training error and test error are low for the model. A lower polynomial increases training and test error, whereas a higher polynomial exponentially increases the test error. This is clear evidence of overfitting.

For a polynomial of degree $m = 6$ and higher, the test error increases exponentially whereas the training error decreases slightly. The model is overfitting the training data i.e. it is matching '*too perfectly*' with the training data, thereby reducing the training error. However, if the same model is applied to the test dataset, it is unable to distinguish certain datapoints because of the rigidity of the regression. This is capped at $m = 10$, where the RMS error for training data is lower than 0.25 whereas the RMS error for test data is over 2. This is a degree of N – 1, which generalizes poorly to the test dataset due to overfitting.