Ethan Wong
CS146
2/2/22
Dis 1B

CS 146 PS2 Perceptron and Regression

1. a) OR : Perceptron Exists

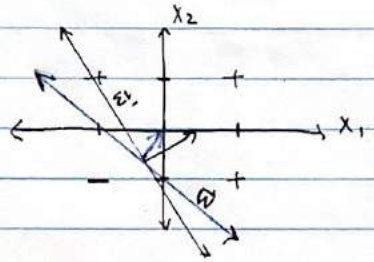
x_1	x_2	y
-1	1	1
-1	-1	-1
1	1	1
1	-1	1

$a = w^T x + b$

$\theta = (\theta_0, \theta_1, \theta_2)$

$x = (1, x_1, x_2)$

$y = \text{sign}(\theta^T x)$



$$\vec{\theta} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \quad \begin{matrix} 1 - 1 + 1 = 1 \\ 1 - 1 - 1 = -1 \\ 1 + 1 + 1 = 3 \\ 1 - 1 + 1 = 1 \end{matrix} \qquad \vec{\theta}' = \begin{pmatrix} 2 \\ 2 \\ 3 \end{pmatrix} \quad \begin{matrix} 2 - 2 + 3 = 3 \\ 2 - 2 - 3 = -3 \\ 2 + 2 + 3 = 7 \\ 2 + 2 - 3 = 1 \end{matrix}$$

$\theta_0 + \theta_1 x_1 + \theta_2 x_2$

$\theta_0 - \theta_1 + \theta_2 = +$ $\quad \theta_0 = \theta_1 = 1$

$\theta_0 - \theta_1 - \theta_2 = -$ $\quad -\theta_2 = -$

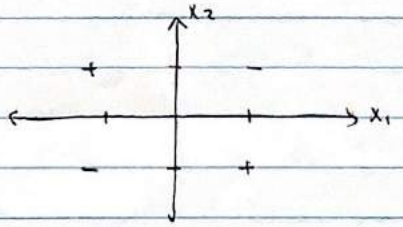
$2\theta_0 - 2\theta_1 = 0$ $\quad +\theta_2 = +$

$\theta_0 - \theta_1 = 0$ $\quad \theta_2 = 1$

$\boxed{(1, 1, 1)}$

b) XOR : Perceptron does not exist

x_1	x_2	y
-1	1	1
-1	-1	-1
1	1	-1
1	-1	1



No perceptron exists for XOR as the data is not linearly separable, which we see in the plot above.

(2)

$$J(\theta) = -\sum_{n=1}^{N} [y_n \log h_\theta(x_n) + (1-y_n)\log(1-h_\theta(x_n))]$$

2. a) $$\frac{\partial J}{\partial \theta_j} = -\sum_{n=1}^{N} \frac{\partial [y_n \log h_\theta(x_n)]}{\partial \theta_j} + \frac{\partial [(1-y_n)\log(1-h_\theta(x_n))]}{\partial \theta_j}$$

$$\frac{\partial J}{\partial \theta_j} = -\sum_{n=1}^{N} y_n \frac{\partial [\log h_\theta(x_n)]}{\partial \theta_j} + (1-y_n)\frac{\partial [\log(1-h_\theta(x_n))]}{\partial \theta_j}$$

$$h_\theta(x) = \sigma(\theta^T x)$$

$$\frac{\partial h_\theta(x)}{\partial \theta_K} = \frac{\partial \sigma(\theta^T x)}{\partial \theta_K}$$

$$= \frac{\partial \sigma(\theta^T x)}{\partial \theta^T x} \frac{\partial (\theta^T x)}{\partial \theta_K}$$

$$= \sigma(\theta^T x)(1-\sigma(\theta^T x)) \frac{\partial (\sum_j \theta_j x_j)}{\partial \theta_K}$$

$$= \sigma(\theta^T x)(1-\sigma(\theta^T x)) x_K$$

$$= h_\theta(x)(1-h_\theta(x)) x_K$$

$$\frac{\partial \log h_\theta(x_n)}{\partial \theta_j} = \frac{\partial \log h_\theta(x_n)}{\partial h_\theta(x_n)} \frac{\partial h_\theta(x_n)}{\partial \theta_j} = \frac{1}{h_\theta(x_n)} h_\theta(x_n)(1-h_\theta(x_n)) x_j = (1-h_\theta(x_n)) x_j$$

$$\frac{\partial \log(1-h_\theta(x_n))}{\partial \theta_j} = -h_\theta(x_n) x_j$$

$$\frac{\partial J}{\partial \theta_j} = -\sum_{n=1}^{N} (y_n(1-h_\theta(x_n))x_j - (1-y_n)h_\theta(x_n)x_j)$$

$$\frac{\partial J}{\partial \theta_j} = -\sum_{n=1}^{N} (y_n - h_\theta(x_n)) x_{n,j}$$

$$\boxed{\frac{\partial J}{\partial \theta_j} = \sum_{n=1}^{N} (h_\theta(x_n) - y_n) x_{n,j}}$$

b) $$\frac{\partial^2 J}{\partial \theta_j \partial \theta_K} = \frac{\partial}{\partial \theta_j}\left(\frac{\partial J}{\partial \theta_K}\right) = \frac{\partial}{\partial \theta_j}\left(\sum_{n=1}^{N} (h_\theta(x_n) - y_n) x_{n,K}\right)$$

$$= \sum_{n=1}^{N} \frac{\partial [(h_\theta(x_n) - y_n) x_{n,k}]}{\partial \theta_j}$$

$$= \sum_{n=1}^{N} \frac{\partial [h_\theta(x_n) x_{n,k}]}{\partial \theta_j} \rightarrow \sum_{n=1}^{N} \frac{\partial h_\theta(x_n)}{\partial \theta_j} x_{n,k}$$

$$\boxed{\frac{\partial^2 J}{\partial \theta_j \partial \theta_K} = \sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n)) x_{n,j} x_{n,K}}$$

Hessian $H = \sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))\, x_n x_n^T$

$H_{j,k} = \sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))(x_n x_n^T)_{j,k}$

$H_{j,k} = \underbrace{\sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))\, x_{n,j} x_{n,k}}_{\frac{d^2 j}{d\theta_j d\theta_k}}$ (on bottom of prev. page)

Thus, the Hessian can be written as H.

c) $z^T H z = \sum_{j,k} z_j z_k H_{jk} \geq 0$

$z^T H z = z^T \left(\sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))\, x_n x_n^T\right) z$

$= \sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))\, x_n x_n^T z^T z$ → [$X = xx^T$ iff $X_{i,j} = x_i x_j$]

$= \sum_{n=1}^{N} \underbrace{h_\theta(x_n)}_{①} \underbrace{(1-h_\theta(x_n))}_{②} \underbrace{(z^T x_n)^2}_{③}$

① : $[0,1]$ ② : $[0,1]$ ③ : Greater than 0 (≥ 0)

$(z^T x_n)^2 \geq 0 \quad 0 \leq h_\theta(x_n) \leq 1 \quad 0 \leq (1-h_\theta(x_n)) \leq 1$

This means that the expression

$z^T H z = \sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))(z^T x_n)^2 \geq 0$

Thus, $H \succeq 0$.

3. a) $P(X_i = 1) = \theta \qquad P(X_i = 0) = 1 - \theta$

$\hat{\theta}_{MLE} = \underset{\theta}{\operatorname{argmax}}\, L(\theta)$

Given the information above:

$P(X_i; \theta) = \theta^{X_i}(1-\theta)^{(1-X_i)}$

Likelihood Function: $L(\theta) = \prod_{i=1}^{n} P(X_i; \theta) = \prod_{i=1}^{n} \theta^{X_i}(1-\theta)^{(1-X_i)}$

- The likelihood function does not actually depend on the order in which the random variables are observed. This is because we are told that the given values X_i are independent variables from the same Bernoulli distribution with parameter θ. Because of this, we know they are I I D. (independent and identically distributed random variables). Knowing this, we can conclude that the likelihood is simply the product of independent probability.

b) $\ell(\theta) = \log(L(\theta)) = \sum_n \log\left[\theta^{x_i}(1-\theta)^{(1-x_i)}\right]$

$\ell(\theta) = \sum_n x_i \log\theta + (1-x_i)\log(1-\theta)$

$\ell'(\theta) = \sum_n \frac{x_i}{\theta} - \frac{1-(x_i)}{1-\theta}$

$\ell''(\theta) = \sum_n \frac{-x_i}{\theta^2} - \frac{-(1-x_i)(-1)}{(1-\theta)^2}$

$= \sum_n \frac{-x_i}{\theta^2} - \frac{(1-x_i)}{(1-\theta)^2}$

$= -\sum_n \frac{x_i}{\theta_2} + \frac{(1-x_i)}{(1-\theta)^2}$

$$\ell(\theta) = \sum_n x_i \log\theta + (1-x_i)\log(1-\theta) \qquad \ell'(\theta) = \sum_n \frac{x_i}{\theta} - \frac{(1-x_i)}{1-\theta}$$

$$\ell''(\theta) = -\sum_n \frac{x_i}{\theta^2} + \frac{(1-x_i)}{(1-\theta)^2}$$

We see that $\theta \in [0,1]$ and $x_i \in [0,1]$. This means that the second derivative is negative for all values of θ. This means that the log-likelihood function will concave down - the critical point would thus be the maximum, which is the MLE.

$$\ell'(\theta) = \sum_n \left[\frac{x_i}{\theta} - \frac{(1-x_i)}{1-\theta}\right] = 0$$

$$0 = \sum_n \theta(1-\theta)\left[\frac{x_i}{\theta} - \frac{(1-x_i)}{1-\theta}\right]$$

$$\sum_n (x_i(1-\theta) - (1-x_i)\theta) = 0$$

$$\sum_n (x_i - x_i\theta - \theta + x_i\theta) = 0$$

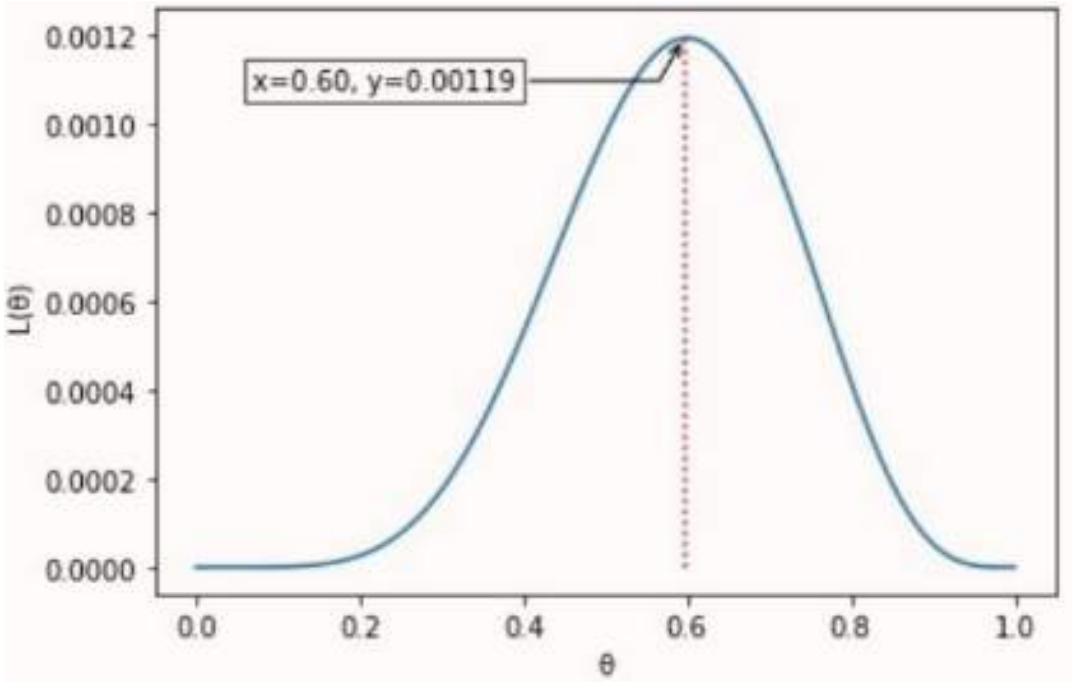
$$\sum_n (x_i - \theta) = 0$$

$$\sum_n x_i - \theta_n = 0$$

$$\theta_n = \sum_n x_i$$

$$\theta = \frac{1}{n}\sum_n x_i$$

3c)



n=10 dataset: six 1's, four 0's

pictured above is the likelihood function for the dataset.

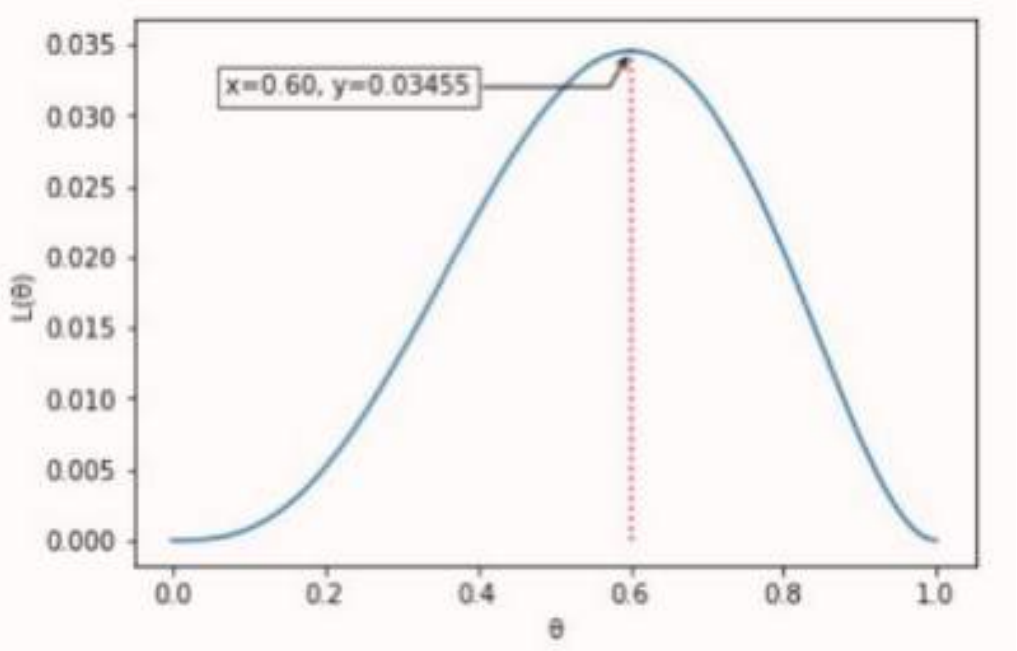
θ_{MLE} is roughly $\theta = 0.6$, according to the plot.

When we use the closed-form function with $n = 10$, we see that

$\theta = \frac{1}{10} \sum_{i=1}^{10} x = \frac{6}{10} = 0.6$

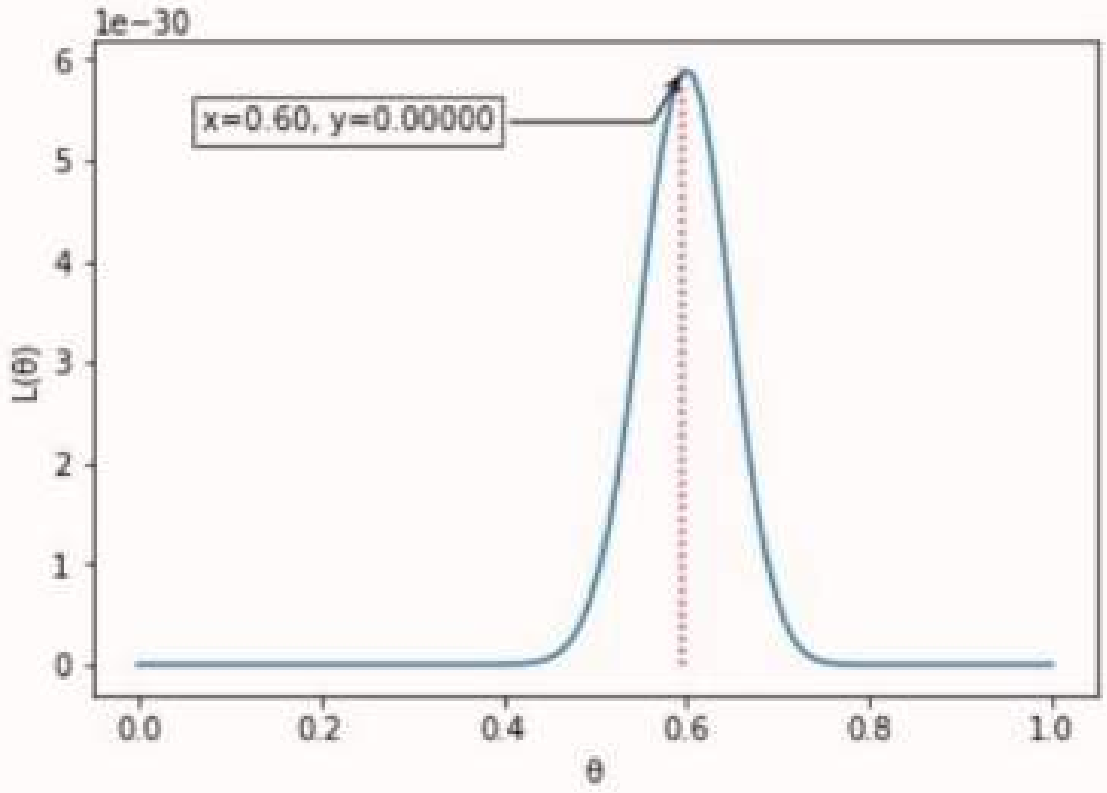
Thus, the answer agrees with the closed form answer.

3d)



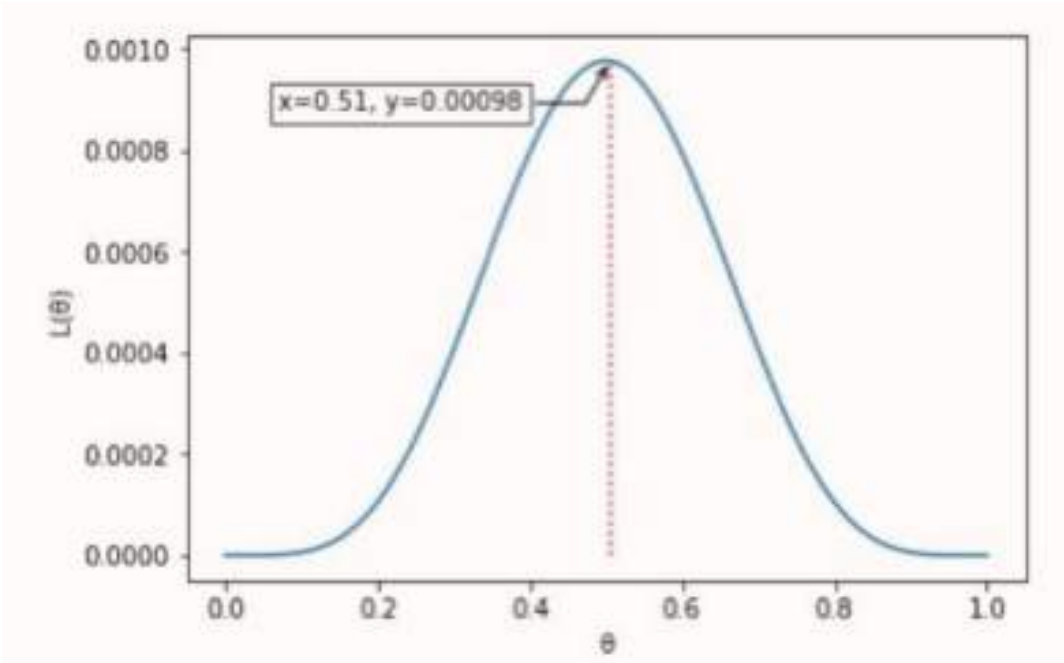
$n = 5$ dataset: 3 1's, 2 0's

pictured above is the likelihood function for the dataset. Now that the dataset is smaller, the likelihood function shows more variance. $\hat{\theta}_{MLE}$ is still roughly $\theta = 0.6$, according to the plot. However, the likelihood $L(\theta)$ is now at a maximum of $L(\theta) = 0.035$, whereas in part c it was $L(\theta) = 0.00119$ with a bigger data set. The curve is also much wider now.



$n = 100$ dataset : 60 1's, 40 0's

pictured above is the likelihood function for the dataset. Now that the dataset is much larger, the likelihood function shows less variance. $\hat{\theta}_{MLE}$ is still roughly $\theta = 0.6$, according to the plot. However, the likelihood $L(\theta)$ is now at a miniscule amount, roughly $6e^{-30}$. The curve is much skinnier now.



$n = 10$ dataset: 5 1's, 5 0's

pictured above is the likelihood function of the dataset

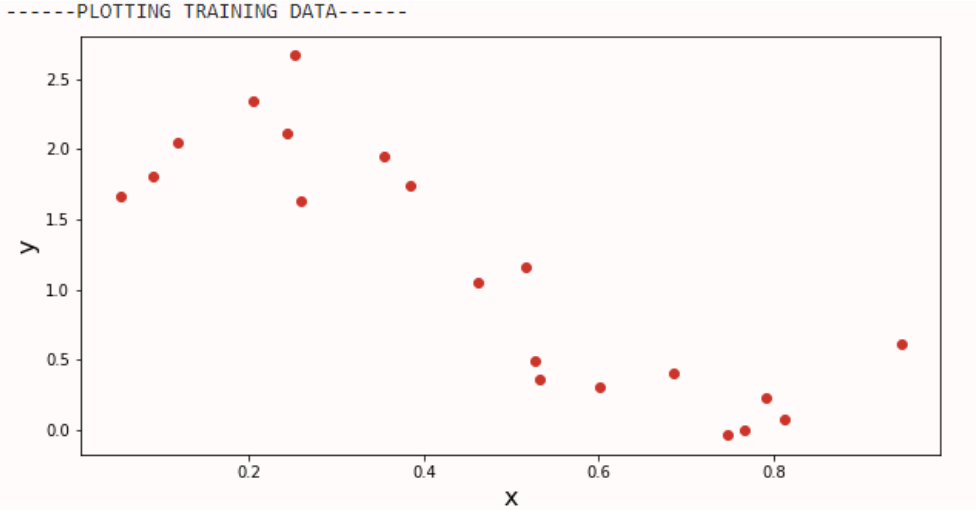
This dataset is the same size, but has a different sample mean. $\hat{\theta}_{MLE}$ has now changed to roughly $\theta = 0.51$. This matches what we expect from the closed form answer:

$$\theta = \frac{5+0}{10} = 0.50$$

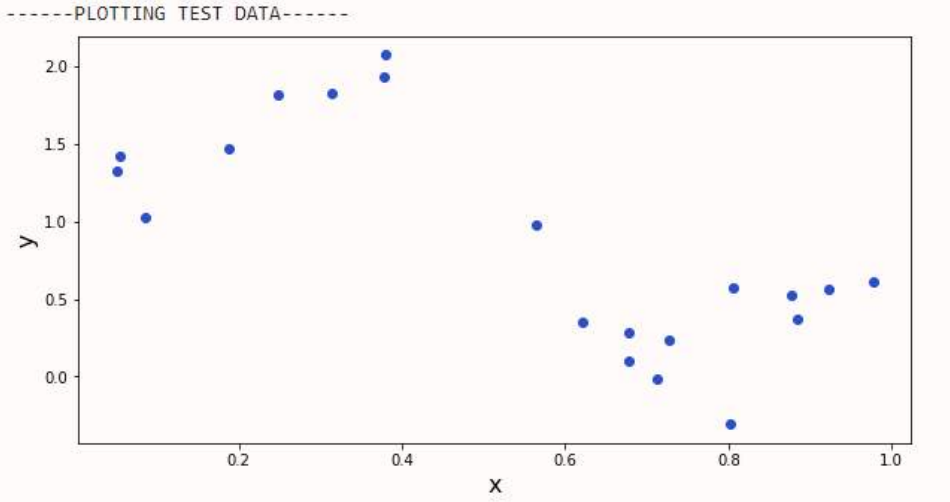
This plot shows an almost identical variance to the dataset in part c. The maximum likelihood is now 0.00098, which is slightly smaller than the maximum likelihood in part c, $L(\theta) = 0.00119$.

4a)

```
------PLOTTING TRAINING DATA------
```




```
------PLOTTING TEST DATA------
```




```
### ========== TODO : START ========== ###
# part a: main code for visualizations
print('Visualizing data...')
print("------PLOTTING TRAINING DATA------")
plot_data(train_data.X, train_data.y, color='#cc362b')
print("------PLOTTING TEST DATA------")
plot_data(test_data.X, test_data.y, color='#2b50cc')
### ========== TODO : END ========== ###
```

Neither the training dataset nor the test dataset exhibit a strong linear relationship between their data points. This means that linear regression is unlikely to be the best option for this dataset; it will likely make a lot of poor predictions that won't be helpful. The data seems to resemble a more polynomial relationship, so it would be smarter to use polynomial regression as it will make more accurate predictions.

4b) “You do not need to turn in anything for this part.”

4c) “You do not need to turn in anything for this part.”

4d)

First code snippet

```
### ========== TODO : START ========== ###
# part d: update theta (self.coef_) using one step of GD
# hint: you can write simultaneously update all theta using vector math
XTX = np.dot(X.T, X)  # matrix multiplication
XTXtheta = np.dot(XTX, self.coef_)  # matrix multiply with coefficient
XTY = np.dot(X.T, y)  # matrix multiplication
grad = XTXtheta - XTY  # calculate the gradient
self.coef_ = np.subtract(self.coef_, 2 * eta * grad) # update theta based on calculation
# track error
# hint: you cannot use self.predict(...) to make the predictions
y_pred = np.dot(X, self.coef_)
err_list[t] = np.sum(np.power(y - y_pred, 2)) / float(n)
### ========== TODO : END ========== ###
```

Second code snippet

```
def cost(self, X, y) :
    """
    Calculates the objective function.

    Parameters
    --------------------
        X       -- numpy array of shape (n,d), features
        y       -- numpy array of shape (n,), targets

    Returns
    --------------------
        cost    -- float, objective J(theta)
    """
    ### ========== TODO : START ========== ###
    # part d: compute J(theta)
    cost = np.sum(np.power(y - self.predict(X), 2))
    ### ========== TODO : END ========== ###
    return cost
```

When I ran the code from above, I found a cost of 40.233847409671, which rounds to 40.234. This matches the value that was mentioned in the problem set spec, so it seems that everything ran correctly.

Learning Rate (**η**)	Coefficients	# of Iterations	Final Cost Value
10^{-6}	[0.36400847; 0.09215787]	10,000	25.86329625891011
10^{-5}	[1.15699657; -0.22522908]	10,000	13.158898555756045
10^{-3}	[2.44640682; -2.81635304]	7,055	3.9125764057918775
0.05	[nan nan]	10,000	nan

It seems that when $\eta=10^{-6}$ or $\eta=10^{-5}$, the algorithm does not converge. The fact that there were 10,000 iterations means that we did not have to terminate the algorithm early — meaning that it never converged. We can also tell that the algorithm did not converge for $\eta=10^{-6}$ or $\eta=10^{-5}$ because the final cost value was still quite large, especially compared to $\eta=10^{-3}$, which did converge. We know that the $\eta=10^{-3}$ converged because the number of iterations is 7,055, which is much less than 10,000.

When using $\eta=0.05$, something strange happens. The coefficients we observe are [nan; nan] (nan meaning Not A Number). We see that the final cost value is also nan. There are also some warnings that show up when using $\eta=0.05$ (see below). The reason that this happens is because when you select a step size that is too large, the algorithm will also not converge, similarly to what we saw in $\eta=10^{-6}$ or $\eta=10^{-5}$. The computations do not work correctly which is why we get nan in the results — there is overflow occurring here.

```
/usr/local/lib/python3.7/dist-packages/numpy/core/fromnumeric.py:87: RuntimeWarning: overflow encountered in reduce
  return ufunc.reduce(obj, axis, dtype, out, **passkwargs)
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:110: RuntimeWarning: invalid value encountered in subtract
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:106: RuntimeWarning: overflow encountered in power
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:102: RuntimeWarning: invalid value encountered in subtract
number of iterations: 10000
----------------------------------------
Learning Rate:  0.05
Coefficients:  [nan nan]
Cost:  nan
```

Using these different values shows that it is very important to select an appropriate step size for the gradient descent algorithm to run efficiently. Using a step size that is too small or too big can cause the algorithm to converge very slowly, or not converge at all.

4e)

```
def fit(self, X, y) :
    """
    Finds the coefficients of a {d-1}^th degree polynomial
    that fits the data using the closed form solution.

    Parameters
    --------------------
        X       -- numpy array of shape (n,d), features
        y       -- numpy array of shape (n,), targets

    Returns
    --------------------
        self    -- an instance of self
    """

    X = self.generate_polynomial_features(X) # map features

    ### ========== TODO : START ========== ###
    # part e: implement closed-form solution
    # hint: use np.dot(...) and np.linalg.pinv(...)
    #       be sure to update self.coef_ with your solution
    temp  = np.dot(X.T, X)
    XTX = np.linalg.pinv(np.dot(X.T, X))
    XTY = np.dot(X.T, y)
    self.coef_ = np.dot(XTX, XTY)
    ### ========== TODO : END ========== ###
```

The closed form coefficients are: 2.44640709; -2.81635359]
The cost is: 3.9125764057914636

Both the values for the coefficients and the cost are almost identical to the values seen in part d, where we used $\eta=10^{-3}$. A benefit of the closed form algorithm is that it executes quicker than gradient descent because it only has to compute matrix computations. Gradient descent tends to be slower because it has to perform matrix computations over many iterations until it converges (or doesn't), which can take quite a long time.

4f)

```
### ========== TODO : START ========== ###
# part f: update step size
# change the default eta in the function signature to 'eta=None'
# and update the line below to your learning rate function
if eta_input is None :
    eta = 1/(1+t)
else :
    eta = eta_input
### ========== TODO : END ========== ###
```

When we change the learning rate to be a function of k where $\eta_k=1/(1+k)$, we observe the following:
Number of iterations: 1,350
The coefficients are: [2.44640744; -2.81635429]
The cost is: 3.912576405792132

4g) “You do not need to turn in anything for this part.”

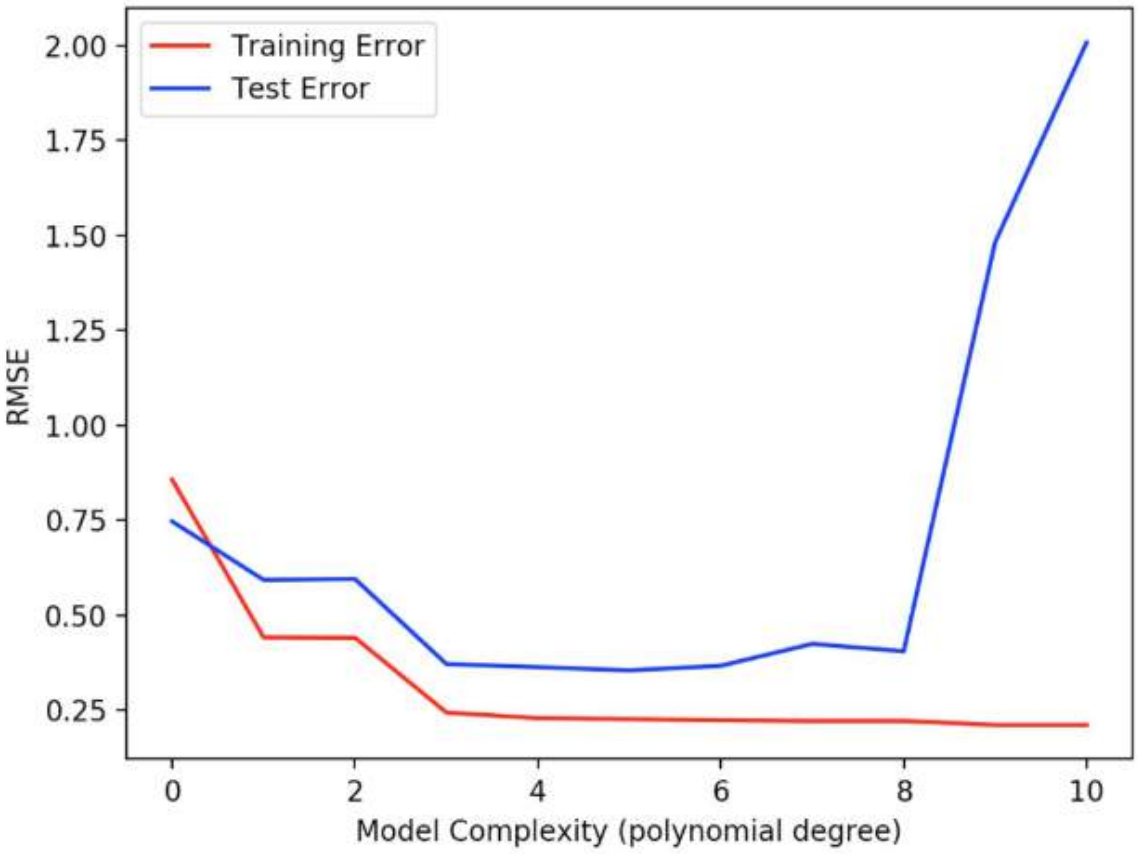
4h)

```
def rms_error(self, X, y) :
    """
    Calculates the root mean square error.

    Parameters
    --------------------
        X       -- numpy array of shape (n,d), features
        y       -- numpy array of shape (n,), targets

    Returns
    --------------------
        error   -- float, RMSE
    """
    ### ========== TODO : START ========== ###
    # part h: compute RMSE
    N, d = X.shape
    error = np.sqrt(self.cost(X, y)/N)
    ### ========== TODO : END ========== ###
    return error
```

In general, we tend to prefer RSME as a metric over J(θ) because RSME takes into account the set size and normalizes J(θ) using n instances. This means that the final result is more general — it can be generalized to other data sets and is also more comparable across data sets of different sizes. In addition, we know that MSE is a representative of the sample variance and RMS is a representative of the sample standard deviation of the residuals. This means that RMS is more useful to us than MSE is because it would be on the same scale as our predictions.

To put it simply, RSME is better at indicating error per example. This means that our error comparisons are less susceptible to error biases from the size of a data set.

4i)



The degree polynomials that best fit the data are model complexity(m) = 3,4,5,6 because these are the values where the training error and test error are the lowest.

When m is less than 3, we observe that there is underfitting because both the training error and test error are high.

On the other hand, as m increases past 6, we observe overfitting starting to occur. The training error is still very low while test error begins to spike (especially at m=8). This indicates that the model is not generalizing well, which is the main sign of overfitting.