CM146, Winter 2021

Problem Set 2: Perceptron and Regression

Due Feb 12, 2021

Name: Joseph Picchi

UID: 605-124-511

Problem 1: Perceptron

(a) Problem 1a

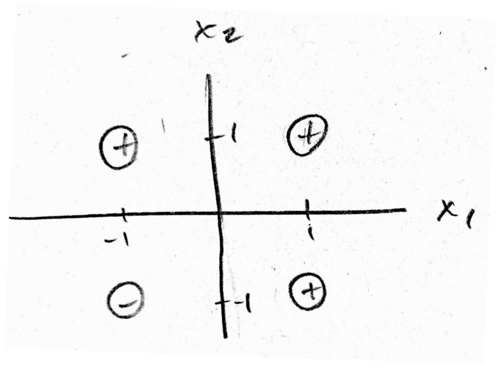
Solution:

For a two input perceptron,

$X = \begin{bmatrix} 1 & x_1 & x_2 \end{bmatrix}^T$

$\theta = \begin{bmatrix} \theta_0 & \theta_1 & \theta_2 \end{bmatrix}^T$

Our dataset can be represented as follows:



One possible perceptron is:

$\theta = [1 \quad 1 \quad 1]^T$

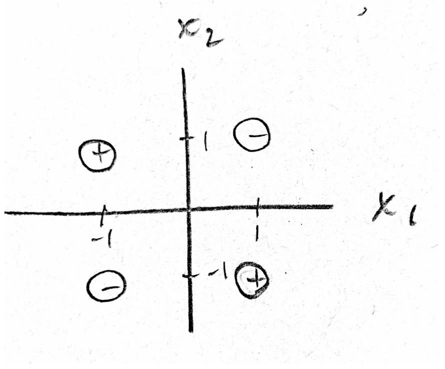
The solution is not unique because another possible perceptron is:

$\theta = [1.1 \quad 1 \quad 1]^T$

(b) Problem 1b

Solution:

Our dataset can be represented as follows:



No perceptron exists because the data is not linearly separable in $\mathbb{R}^2$.

Problem 2: Logistic Regression

(a) Problem 2a

Solution:

For this problem, I assume that $\log$ is of base e, as was assumed in lecture.

I first calculate the components of the solution, then put them together at the end to generate the final answer for $\frac{\partial J}{\partial \theta_j}$.

Notational note: $x_n = \begin{bmatrix} 1 & x_{n1} & x_{n2} & \dots & x_{nD} \end{bmatrix}^T$

$$\begin{aligned}\frac{\partial a}{\partial \theta_i} &= \frac{\partial}{\partial \theta_i}\left[\theta^T x_n\right] \\ &= \frac{\partial}{\partial \theta_i}\left[\theta_0 + \theta_1 x_{n1} + \ldots + \theta_j x_{nj} + \ldots + \theta_D x_{nD}\right] \\ &= x_{nj}\end{aligned}$$

$$
\begin{aligned}
\frac{\partial \sigma}{\partial a} &= \frac{\partial}{\partial a}\left[\sigma(a)\right] \\
&= \frac{\partial}{\partial a}\left[\frac{1}{1+e^{-a}}\right] \\
&= -\left[1+e^{-a}\right]^{-2} \cdot \frac{\partial}{\partial a}\left[1+e^{-a}\right] \\
&= -\left[1+e^{-a}\right]^{-2}(-e^{-a}) \\
&= \frac{1}{(1+e^{-a})^2}(e^{-a}) \\
&= \frac{1}{1+e^{-a}} \cdot \frac{e^{-a}}{1+e^{-a}} \\
&= \sigma(a) \cdot \left(\frac{1+e^{-a}}{1+e^{-a}} - \frac{1}{1+e^{-a}}\right) \\
&= \sigma(a)\big(1-\sigma(a)\big)
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial J}{\partial \theta_j} &= \frac{\partial}{\partial \theta_j}\left[-\sum_{n=1}^{N}\left[y_n \log h_\theta(x_n) + (1-y_n)\log(1-h_\theta(x_n))\right]\right] \\
&= -\sum_{n=1}^{N}\frac{\partial}{\partial \theta_j}\left[y_n \log \sigma(a) + (1-y_n)\log(1-\sigma(a))\right] \\
&= -\sum_{n=1}^{N}\left[(y_n)\left(\frac{1}{\sigma(a)}\right)\left(\frac{\partial \sigma}{\partial a}\right)\left(\frac{\partial a}{\partial \theta_j}\right) + (1-y_n)\left(\frac{1}{1-\sigma(a)}\right)\left(-\frac{\partial \sigma}{\partial a}\right)\left(\frac{\partial a}{\partial \theta_j}\right)\right] \\
&= -\sum_{n=1}^{N}\left[(y_n)\left(\frac{1}{\sigma(a)}\right)\cdot \sigma(a)(1-\sigma(a)) \cdot x_{nj} + (y_n-1)\left(\frac{1}{1-\sigma(a)}\right)\cdot \sigma(a)(1-\sigma(a)) \cdot x_{nj}\right] \\
&= -\sum_{n=1}^{N}\left[y_n\left(1-\sigma(a)\right)x_{nj} + (y_n-1)\cdot\sigma(a)\cdot x_{nj}\right] \\
&= -\sum_{n=1}^{N} x_{nj}\left[y_n - y_n\sigma(a) + y_n\sigma(a) - \sigma(a)\right] \\
&= \sum_{n=1}^{N} x_{nj}\left[\sigma(a) - y_n\right] \\
&= \sum_{n=1}^{N} x_{nj}\left[h_\theta(x_n) - y_n\right]
\end{aligned}
$$

(b) Problem 2b

Solution:

$$\begin{aligned}
\frac{\partial^2 J}{\partial\theta_j\partial\theta_k} &= \frac{\partial}{\partial\theta_j}\left(\frac{\partial J}{\partial\theta_k}\right)\\
&= \frac{\partial}{\partial\theta_j}\left(\sum_{n=1}^{N} x_{nk}\left[\sigma(a)-y_n\right]\right)\\
&= \sum_{n=1}^{N} x_{nk}\cdot\frac{\partial}{\partial\theta_j}\left[\sigma(a)-y_n\right]\\
&= \sum_{n=1}^{N} x_{nk}\cdot\frac{\partial\sigma}{\partial a}\cdot\frac{\partial a}{\partial\theta_j}\\
&= \sum_{n=1}^{N} x_{nk}\cdot\sigma(a)(1-\sigma(a))\cdot x_{nj}\\
&= \sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))\cdot x_{nk}x_{nj}
\end{aligned}$$

$$H = \begin{bmatrix}
\frac{\partial^2 J}{\partial\theta_1^2} & \frac{\partial^2 J}{\partial\theta_1\partial\theta_2} & \cdots & \frac{\partial^2 J}{\partial\theta_1\partial\theta_D}\\
\frac{\partial^2 J}{\partial\theta_2\partial\theta_1} & \frac{\partial^2 J}{\partial\theta_2^2} & \cdots & \frac{\partial^2 J}{\partial\theta_2\partial\theta_D}\\
\vdots & \vdots & \ddots & \vdots\\
\frac{\partial^2 J}{\partial\theta_D^2} & \frac{\partial^2 J}{\partial\theta_D\partial\theta_2} & \cdots & \frac{\partial^2 J}{\partial\partial\theta_D^2}
\end{bmatrix}$$

$$= \begin{bmatrix}
\sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))\cdot x_{n1}^2 & \cdots & \sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))\cdot x_{n1}x_{nD}\\
\vdots & \vdots & \vdots\\
\sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))\cdot x_{nD}x_{n1} & \cdots & \sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))\cdot x_{nD}^2
\end{bmatrix}$$

$$= \sum_{n=1}^{N}\begin{bmatrix}
h_\theta(x_n)(1-h_\theta(x_n))\cdot x_{n1}^2 & \cdots & h_\theta(x_n)(1-h_\theta(x_n))\cdot x_{n1}x_{nD}\\
\vdots & \vdots & \vdots\\
h_\theta(x_n)(1-h_\theta(x_n))\cdot x_{nD}x_{n1} & \cdots & h_\theta(x_n)(1-h_\theta(x_n))\cdot x_{nD}^2
\end{bmatrix}$$

$$= \sum_{n=1}^{N} \left(h_\theta(x_n)(1 - h_\theta(x_n)) \begin{bmatrix} x_{n1}^2 & \cdots & x_{n1}x_{nD} \\ \vdots & \vdots & \vdots \\ x_{nD}x_{n1} & \cdots & x_{nD}^2 \end{bmatrix} \right)$$

$$= \sum_{n=1}^{N} \left(h_\theta(x_n)(1 - h_\theta(x_n)) \right) \begin{bmatrix} x_{n1} \\ x_{n2} \\ \vdots \\ x_{nD} \end{bmatrix} \begin{bmatrix} x_{n1} & x_{n2} & \cdots & x_{nD} \end{bmatrix}$$

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

(c) Problem 2c

Solution:

$$\begin{aligned} z^T H z &= z^T \left(\sum_{n=1}^{N} h_\theta(x_n)(1 - h_\theta(x_n)) x_n x_n^T \right) z \\ &= \sum_{n=1}^{N} h_\theta(x_n)(1 - h_\theta(x_n)) z^T x_n x_n^T z \\ &= \sum_{n=1}^{N} h_\theta(x_n)(1 - h_\theta(x_n)) (z^T x_n)(x_n^T z) \\ &= \sum_{n=1}^{N} h_\theta(x_n)(1 - h_\theta(x_n)) (x_n^T z)^T (x_n^T z) \\ &= \sum_{n=1}^{N} h_\theta(x_n)(1 - h_\theta(x_n)) \left\| x_n^T z \right\|_2^2 \\ &\geq 0 \end{aligned}$$

We know that the second to last line is ≥ 0 because…

- $h_\theta(x_n) = \sigma(\theta x_n) \in [0,1]$, so $h_\theta(x_n) \geq 0$ and $(1 - h_\theta(x_n)) \geq 0$
- The $l2$ norm of a vector is always ≥ 0, so the squared $l2$ norm of a vector is also ≥ 0
- The product $p_1 p_2 \cdots p_i \cdots p_n \geq 0$ if each term $p_i \geq 0$
- $\therefore$ the second to last line is ≥ 0

$H \preccurlyeq 0$ because $z^T H z \geq 0$ for all real vectors z

$\therefore$ J is a convex function

Problem 3: Maximum Likelihood Estimation

(a) Problem 3a

Solution:

For each random variable X_i,

$$P(X_i;\theta) = \begin{cases} \theta & if\ X_i = 1 \\ 1-\theta & if\ X_i = 0 \end{cases}$$

We can equivalently express this as:

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

Thus, the likelihood function is:

$$\begin{aligned} L(\theta) &= P(X_1, \ldots, X_n; \theta) \\ &= P(X_1;\theta)P(X_2;\theta)\cdots P(X_n;\theta) \\ &= \prod_{i=1}^{n} P(X_i;\theta) \\ &= \prod_{i=1}^{n} \theta^{X_i}(1-\theta)^{1-X_i} \end{aligned}$$

The likelihood function does not depend on the order in which the random variables are observed because each variable is independent of all other variables and multiplication is a commutative property.

(b) Problem 3b

Solution:

$$
\begin{aligned}
l(\theta) &= \log\big(L(\theta)\big) \\
&= \log\left(\prod_{i=1}^{n}\theta^{X_i}(1-\theta)^{1-X_i}\right) \\
&= \log\theta^{X_1}(1-\theta)^{1-X_1} + \ldots + \log\theta^{X_n}(1-\theta)^{1-X_n} \\
&= \sum_{i=1}^{n}\log\theta^{X_i}(1-\theta)^{1-X_i} \\
&= \sum_{i=1}^{n}\left[\log\theta^{X_i} + \log(1-\theta)^{1-X_i}\right] \\
&= \sum_{i=1}^{n}\left[X_i\log\theta + (1-X_i)\log(1-\theta)\right]
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial l}{\partial\theta} &= \frac{\partial}{\partial\theta}\sum_{i=1}^{n}\left[X_i\log\theta + (1-X_i)\log(1-\theta)\right] \\
&= \sum_{i=1}^{n}\left[X_i\frac{\partial}{\partial\theta}\log\theta + (1-X_i)\frac{\partial}{\partial\theta}\log(1-\theta)\right] \\
&= \sum_{i=1}^{n}\left[X_i\frac{1}{\theta} + (1-X_i)\frac{1}{1-\theta}(-1)\right] \\
&= \sum_{i=1}^{n}\left[\frac{X_i}{\theta} + \frac{X_i-1}{1-\theta}\right] \\
&= \sum_{i=1}^{n}\left[\frac{X_i-\theta X_i}{\theta(1-\theta)} + \frac{\theta X_i-\theta}{\theta(1-\theta)}\right] \\
&= \sum_{i=1}^{n}\left[\frac{X_i-\theta}{\theta(1-\theta)}\right] \\
&= \frac{1}{\theta(1-\theta)}\sum_{i=1}^{n}\left[X_i-\theta\right] \\
&= \frac{1}{\theta(1-\theta)}\left[-n\theta + \sum_{i=1}^{n}X_i\right] \\
&= \frac{-n\theta + \sum_{i=1}^{n}X_i}{\theta(1-\theta)}
\end{aligned}
$$

$$\begin{aligned}
\frac{\partial^2 l}{\partial \theta^2} &= \frac{\partial}{\partial \theta}\left(\frac{\partial l}{\partial \theta}\right) \\
&= \frac{\partial}{\partial \theta}\left(\sum_{i=1}^{n}\left[\frac{X_i}{\theta} + \frac{X_i - 1}{1-\theta}\right]\right) \\
&= \sum_{i=1}^{n}\left[-\frac{X_i}{\theta^2} - \frac{X_i - 1}{(1-\theta)^2}(-1)\right] \\
&= \sum_{i=1}^{n}\left[-\frac{X_i}{\theta^2} + \frac{X_i - 1}{(1-\theta)^2}\right] \\
&= -\frac{1}{\theta^2}\sum_{i=1}^{n} X_i + \frac{1}{(1-\theta)^2}\sum_{i=1}^{n}\left[X_i - 1\right] \\
&= -\frac{\sum_{i=1}^{n} X_i}{\theta^2} + \frac{-n + \sum_{i=1}^{n} X_i}{(1-\theta)^2}
\end{aligned}$$

Finding the closed form solution:

First, find the value of θ that makes the first derivative equal 0.

$$\begin{aligned}
\frac{\partial l}{\partial \theta} &= 0 \\
\frac{-n\theta + \sum_{i=1}^{n} X_i}{\theta(1-\theta)} &= 0 \\
\left(\frac{1}{n}\right)\left(\frac{-n\theta + \sum_{i=1}^{n} X_i}{\theta(1-\theta)}\right) &= \frac{1}{n}\cdot 0 \\
\frac{-\theta + \bar{x}}{\theta(1-\theta)} &= 0 \\
\theta &= \bar{x}
\end{aligned}$$

Then, check to make sure the second derivative is ≤ 0 at the value of θ that we found.

$$\left.\frac{\partial^2 l}{\partial \theta^2}\right|_{\theta=\bar{x}} = \frac{-\sum_{i=1}^{n} X_i}{\bar{x}^2} + \frac{-n + \sum_{i=1}^{n} X_i}{(1-\bar{x})^2} \leq 0$$

We know that the expression above is ≤ 0 by the following logic:

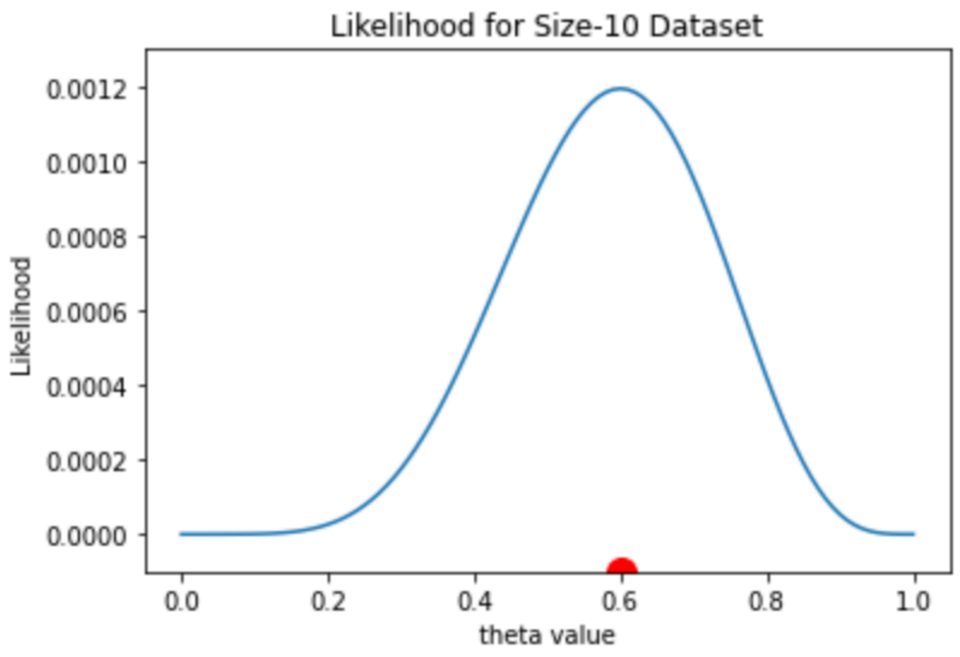
1. Since a squared expression can never be negative, $\bar{x}^2 \geq 0$ and $(1 - \bar{x})^2 \geq 0$
2. Since each $X_i \in \{0,1\}$, each $X_i \geq 0$
3. By (1), $\sum_{i=1}^{n} X_i \geq 0$
4. If each $X_i = 1$, then $\sum_{i=1}^{n} X_i$ achieves its maximum value of $\sum_{i=1}^{n} 1 = n$
5. By (3) and (4), $0 \leq \sum_{i=1}^{n} X_i \leq n$
6. By (3), $-\sum_{i=1}^{n} X_i \leq 0$
7. By (5), $-n + \sum_{i=1}^{n} X_i \leq 0$
8. By (1), (3), and (5), $\frac{-\sum_{i=1}^{n} X_i}{\bar{x}^2} \leq 0$ and $\frac{-n + \sum_{i=1}^{n} X_i}{(1-\bar{x})^2} \leq 0$
9. $\therefore$ the expression above is ≤ 0

At $\theta = \bar{x}$, $\frac{\partial l}{\partial \theta} = 0$ and $\frac{\partial^2 l}{\partial \theta^2} \leq 0$.

$\therefore$ The closed form solution is $\theta = \bar{x}$

(c) Problem 3c

Solution:



The red dot on the x-axis indicates the value of $\hat{\theta}$ that maximizes $L(\theta)$.

Closed form solution: $\hat{\theta} = \bar{x} = \dfrac{6(1) + 4(0)}{10} = 0.6$

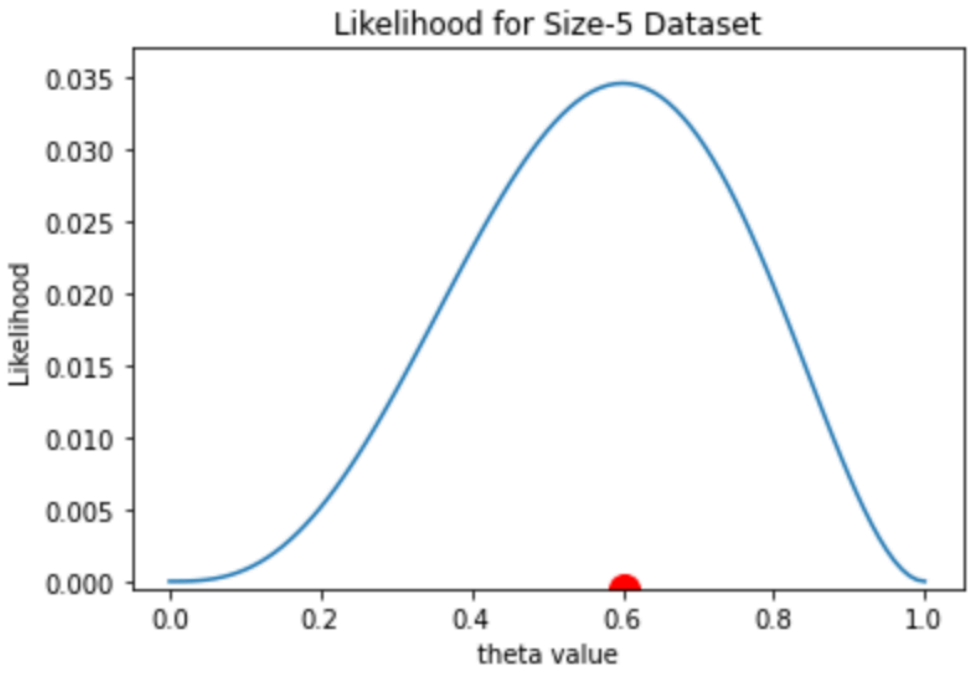
Solution from graph: $\hat{\theta} = 0.6$

$\therefore$ the graph answer agrees with my closed form solution.

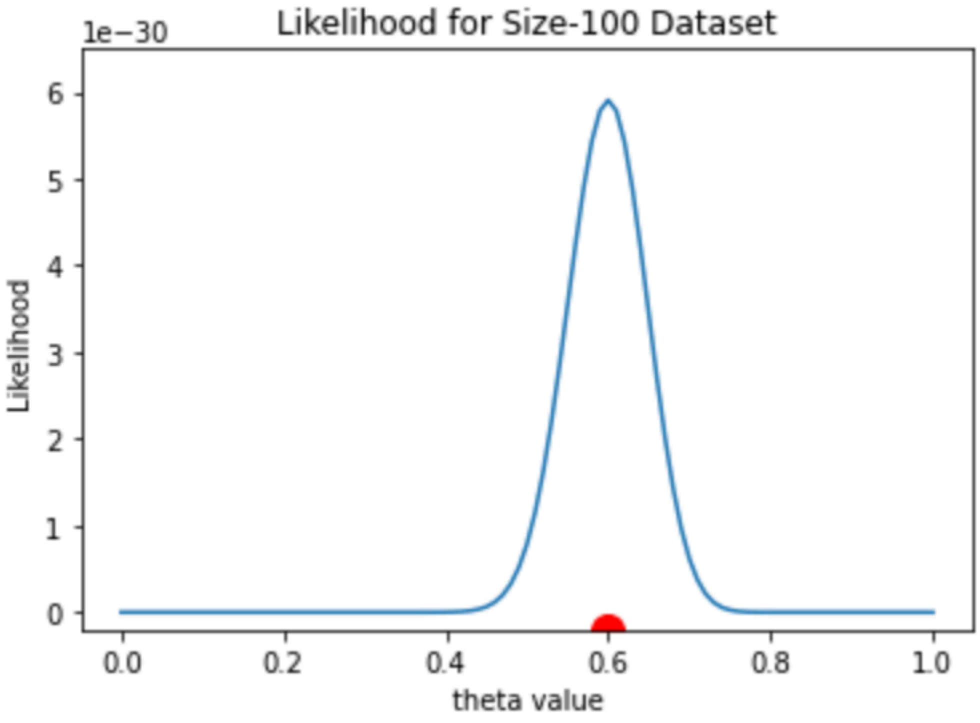
(d) Problem 3d

Solution:

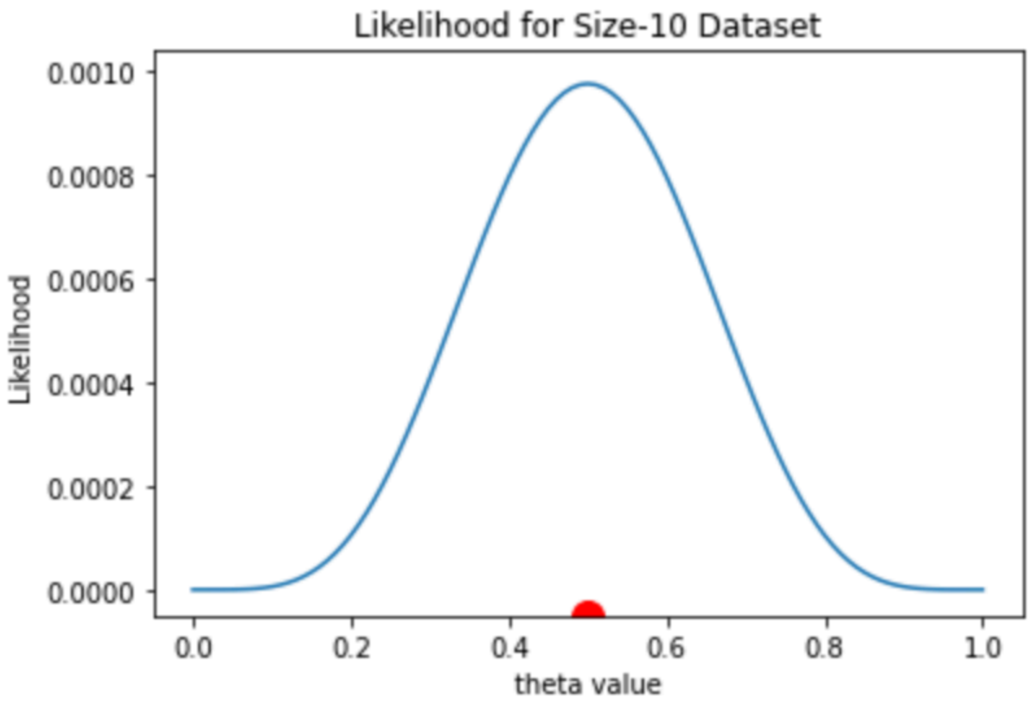
The $n = 5$ likelihood plot:



The $n = 100$ likelihood plot:



The $n = 10$ likelihood plot:



For all 3 plots, the $\hat{\theta}_{MLE}$ is still equivalent to $\bar{x}$, thus verifying my closed form solution. As such, the $\hat{\theta}_{MLE}$ for each of the three plots are:

For n=5, $\hat{\theta}_{MLE} = 0.6$

For n=100, $\hat{\theta}_{MLE} = 0.6$

For n=10, $\hat{\theta}_{MLE} = 0.5$

The main differences between the plots lie in the values for the likelihoods. By looking at the y-axis, we see that the maximum likelihood value decreases by about 1 order of magnitude between n=5 and n=10, and it decreases by about 7 orders of magnitude between n=10 and n=100.

This makes sense because the probability for each X_i is between 0 and 1. Since we multiply together the probability for each X_i in $i = 1, \ldots, n$, the likelihood is continually multiplied by fractional probability values, thus decreasing by orders of magnitude as n increase minimally (eg from n=5 to n=10).

We can explain this intuitively by noting that it is more difficult (ie less probable) to get an exact combination of many values than it is to get an exact combination of only a few values.

Another difference is that larger values of n yield plots with smaller variations. This is evident graphically because the plot for $n = 100$ is more skinny/narrow compared to the plots with lesser values of n. This makes sense because values of $\hat{\theta}$ that differ significantly from $\hat{\theta}_{MLE}$ are more likely to center the mean of the data points at $\hat{\theta}$ as the number of data points increases (a property of the central limit theorem).

Problem 4: Implementation: Polynomial Regression

(a) Problem 4a

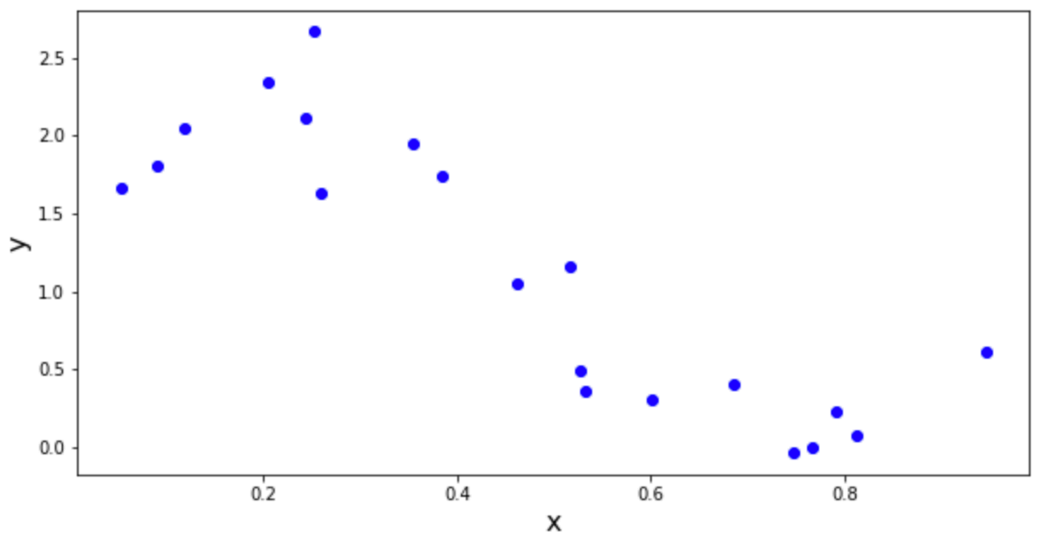
Solution:

Code:

```
### ========== TODO : START ========== ###
# part a: main code for visualizations
print('Visualizing data...')
print("train_data:")
plot_data(train_data.X, train_data.y)
print('test_data:')
plot_data(test_data.X, test_data.y)

### ========== TODO : END ========== ###
```

Training data:

```
train_data:
```



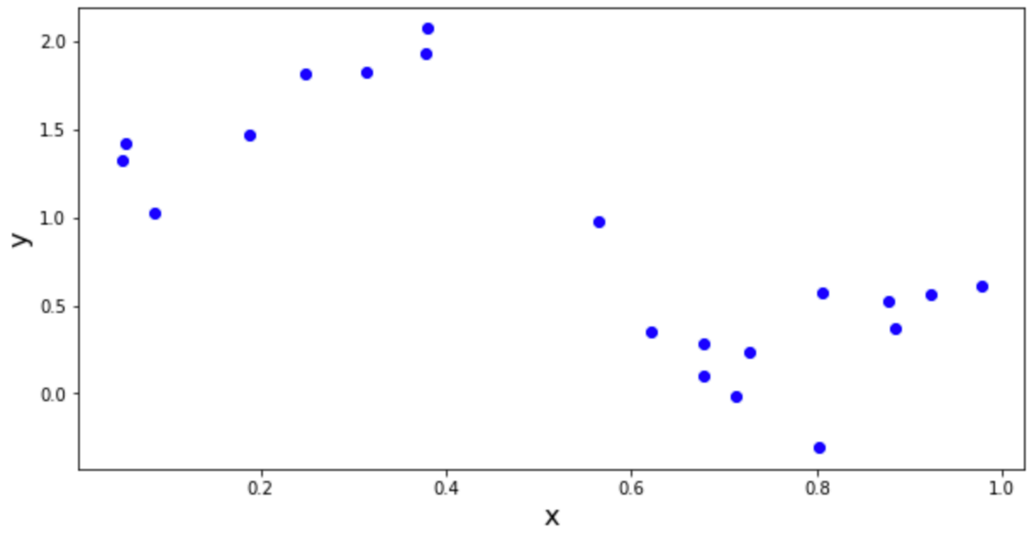
The data seems to follow a straight downward-sloping line in the approximate range of $0.2 \leq x \leq 0.8$, where points to the left of 0.2 and to the right of 0.8 deviate from the straight line formed by the rest of the points.

Linear regression could probably do an adequate job at predicting points in $0.2 \leq x \leq 0.8$ for the aforementioned reasons, though I'd expect slightly more errors when the x value of a point is around ≤ 0.2 or ≥ 0.8.

A higher value of m may cause overfitting because the shape of the points resembles a nonlinear function (e.g. a sin wave).

Test data:

`test_data:`



The points form less of a straight line than they did in the training data, thus suggesting that linear regression will not fit as well compared to the training data, though it may still label the points well enough depending upon the application of the model.

(b) Problem 4b

Solution:

Code for "PolynomialRegression.generate_polynomial_feature(…)"

```
### ========== TODO : START ========== ###
# part b: modify to create matrix for simple linear model
# part g: modify to create matrix for polynomial model
# Phi = X
m = self.m_

Phi = np.empty((n,m+1))
for r in range(0, n):
  # first column of every row has value 1
  Phi[r][0] = 1

  # set the rest of the rows
  for c in range(1, m+1):
    Phi[r][c] = X[r][0]**c

### ========== TODO : END ========== ###
```

(c) Problem 4c

Solution:

Code for “PolynomialRegression.predict(…)”

```
### ========== TODO : START ========== ###
# part c: predict y
y = np.dot(X, self.coef_)
### ========== TODO : END ========== ###
```

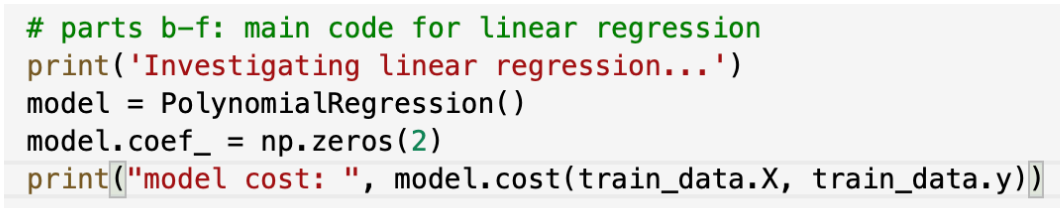
(d) Problem 4d

Solution:

Code for “PolynomialRegression.cost(…)”

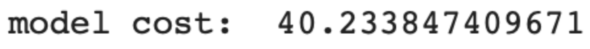
```
### ========== TODO : START ========== ###
# part d: compute J(theta)
squared_errors = (self.predict(X) - y)**2
cost = np.sum(squared_errors)
### ========== TODO : END ========== ###
```

Code snippet to test “PolynomialRegression.cost(…)”:



```
# parts b-f: main code for linear regression
print('Investigating linear regression...')
model = PolynomialRegression()
model.coef_ = np.zeros(2)
print("model cost: ", model.cost(train_data.X, train_data.y))
```

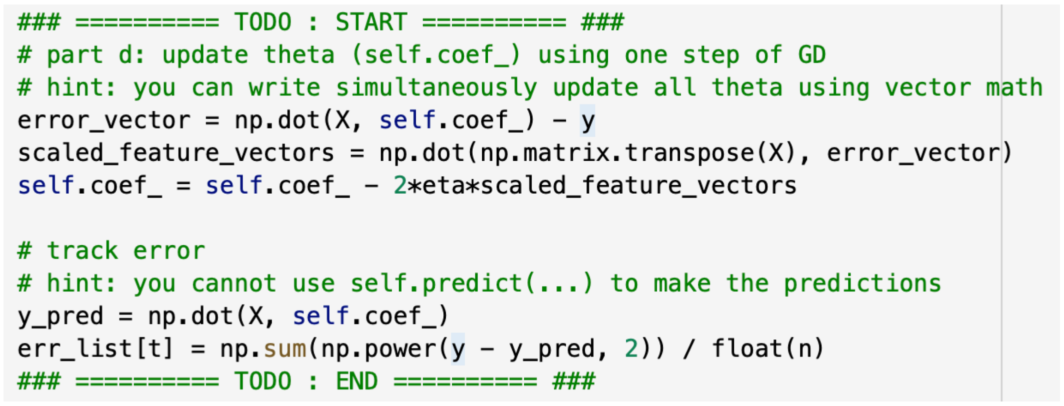
Output of code snippet to test “PolynomialRegression.cost(…)”:



```
model cost:  40.233847409671
```

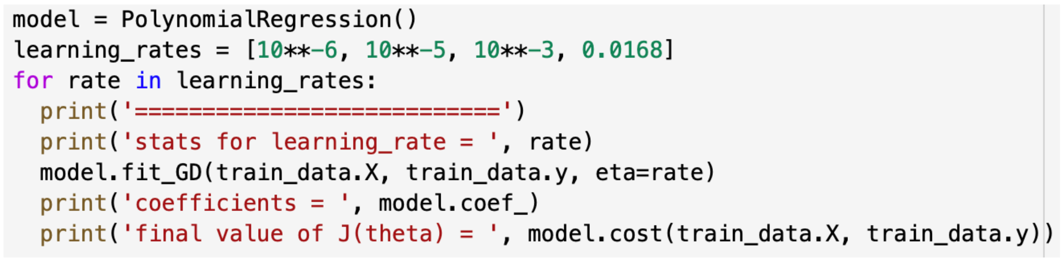
- i.e. the test code snippet produces the correct/expected output

Code for “PolynomialRegression.fit_GD(…)”:



```
### ========== TODO : START ========== ###
# part d: update theta (self.coef_) using one step of GD
# hint: you can write simultaneously update all theta using vector math
error_vector = np.dot(X, self.coef_) - y
scaled_feature_vectors = np.dot(np.matrix.transpose(X), error_vector)
self.coef_ = self.coef_ - 2*eta*scaled_feature_vectors

# 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 ========== ###
```

Code in “main(…)” to experiment with learning rates



```
model = PolynomialRegression()
learning_rates = [10**-6, 10**-5, 10**-3, 0.0168]
for rate in learning_rates:
  print('===========================')
  print('stats for learning_rate = ', rate)
  model.fit_GD(train_data.X, train_data.y, eta=rate)
  print('coefficients = ', model.coef_)
  print('final value of J(theta) = ', model.cost(train_data.X, train_data.y))
```

Output of code in “main(…)” to experiment with learning rates

```
===========================
stats for learning_rate =  1e-06
number of iterations: 10000
coefficients =  [0.36400847 0.09215787]
final value of J(theta) =  25.86329625891011
===========================
stats for learning_rate =  1e-05
number of iterations: 10000
coefficients =  [ 1.15699657 -0.22522908]
final value of J(theta) =  13.158898555756045
===========================
stats for learning_rate =  0.001
number of iterations: 7020
coefficients =  [ 2.4464068 -2.816353 ]
final value of J(theta) =  3.9125764057919463
===========================
stats for learning_rate =  0.0168
number of iterations: 456
coefficients =  [ 2.44640704 -2.81635348]
final value of J(theta) =  3.91257640579148
```

Table of coefficients, number of iterations until convergence, and final value of the objective function:

Learning rate η	10^{-6}	10^{-5}	10^{-3}	0.0168
θ_0	0.36400847	1.15699657	2.4464068	2.44640704
θ_1	0.09215787	-0.22522908	-2.816353	-2.81635348
# iterations	10,000	10,000	7020	456
Final $J(\theta)$	25.8632962589	13.1588985558	3.91257640579	3.91257640579

Coefficient Analysis:

The coefficients for the two η values with the lowest final $J(\theta)$ values (i.e. 10^{-3} and 0.0168) are most similar to each other; both θ_0 and θ_1 are equivalent (disregarding rounding differences) for these two values of η. This makes sense because they have very small $J(\theta)$ values relative to those of the other η values, thus indicating that their coefficients are much closer to the optimal solution compared to 10^{-6} and 10^{-5}.

10^{-6} and 10^{-5} have coefficient values that differ from the others because they are too small, so they failed to approach the optimal solution fast enough and thus were stuck with poorer parameter values after the 10,000 iteration limit.

From the trends in the data, we can also see that the value for θ_0 increases incrementally as the model approaches the optimal θ_0, and the value for θ_1 decreases incrementally as the model approaches the optimal θ_1.

<u>How quickly does each algorithm converge?</u>

The algorithms with lower values of η converged more slowly (i.e. took more iterations to converge), with the two lowest values (i.e. 10^{-6} and 10^{-5}) failing to converge after 10,000 iterations because they did not approach the optimal solution with large enough steps.
Therefore, it makes sense that the number of iterations until convergence is non-increasing from 10,000 to 456 as η increases from 10^{-6} to 0.0168.

<u>Do you notice something strange with $\eta = 0.0168$?</u>

For $\eta = 0.0168$, it makes sense that the number of iterations until convergence is smaller because the learning rate is higher, but it is somewhat strange that the coefficients and $J(\theta)$ values are so close to those of 10^{-3} considering that the learning rate is an order of magnitude greater for $\eta = 0.0168$ relative to 10^{-3}. I would expect that such a large step size might approach the optimal solution stochastically and possibly overshoot the optimal value, but it is possible that the large step size just so happed to land favorably close to the optimal solution its final iterations and subsequently converge there.

(e) Problem 4e

Solution:

Code for "PolynomialRegression.fit(…)":

```
### ========== 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
XTy = np.dot(np.matrix.transpose(X), y)
XTX_inv = np.linalg.pinv(np.dot(np.matrix.transpose(X), X))
self.coef_ = np.dot(XTX_inv, XTy)

### ========== TODO : END ========== ###
```

Code in "main(…)" to analyze coefficients and cost for closed form solution:

```
# testing PolynomialRegression.fit
model.fit(train_data.X, train_data.y)
print('********************** closed form **********************')
print('coefficients for closed form solution = ', model.coef_)
print('final cost for closed form solution = ',
      model.cost(train_data.X, train_data.y))
```

Output of code in “main(…)” to analyze coefficients and cost for closed form solution:

```
********************** closed form **********************
coefficients for closed form solution =  [ 2.44640709 -2.81635359]
final cost for closed form solution =  3.9125764057914636
```

As seen in the output above, the closed form solution coefficients are:

$\theta_0 = 2.44640709$

$\theta_1 = -2.81635359$

The final cost for the closed form solution is:

$J(\theta) = 3.9125764057914636$

The closed form coefficients and cost are equivalent to those obtained by a proper step size (i.e. $\eta = 10^{-3}$ or 0.0168) in GD, barring small rounding differences. They differ in value from the step sizes that were too small in GD (i.e. $\eta = 10^{-5}$ or 10^{-6}) because these latter step sizes were too slow to come adequately close to the optimal solution within 10,000 iterations.

The algorithm runs considerably slower than GD, since, as discussed in lecture, the complexity of the closed form algorithm is $\max\left[O(ND^2), O(D^3)\right]$ and the complexity for batch gradient descent is $O(ND)$. In other words, closed form is slower than GD, and the extent to which it is slower is proportional to the feature count D.

(f) Problem 4f

Solution:

Code for “PolynomialRegression.fit_GD(…)”:

```
### ========== 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 ========== ###
```

Code in “main(…)” to test “PolynomialRegression.fit_GD(…)” with the dynamic learning rate:

```
# testing dynamic learning rate
print('********************** dynamic learning rate **********************')
model.fit_GD(train_data.X, train_data.y)
print('coefficients for dynamic learning rate = ', model.coef_)
print('final cost for dynamic learning rate = ',
      model.cost(train_data.X, train_data.y))
```

Output of code in “main(…)” to test “PolynomialRegression.fit_GD(…)” with the dynamic learning rate:

```
********************** dynamic learning rate **********************
number of iterations: 1731
coefficients for dynamic learning rate =  [ 2.44640672 -2.81635284]
final cost for dynamic learning rate =  3.912576405792244
```

As seen in the output above, the dynamic learning rate causes the model to converge to the same parameters as the closed-form solution (neglecting small rounding differences).

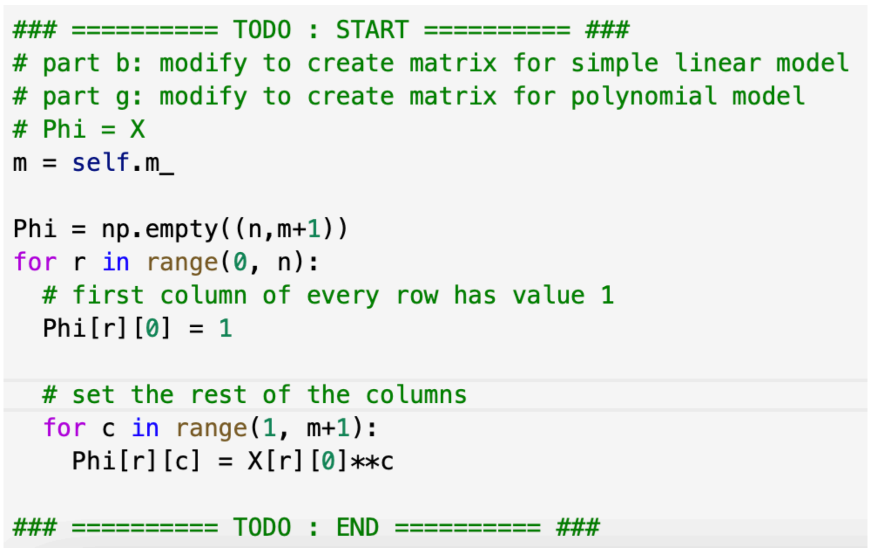
It takes the algorithm 1731 iterations to converge using the dynamic learning rate.

(g) Problem 4g

Solution:

I did not have to update the function because my prior implementation was already generalized to create an $m + 1$ dimensional feature vector for each instance.

Code for “PolynomialRegression.generate_polynomial_features(..)”:

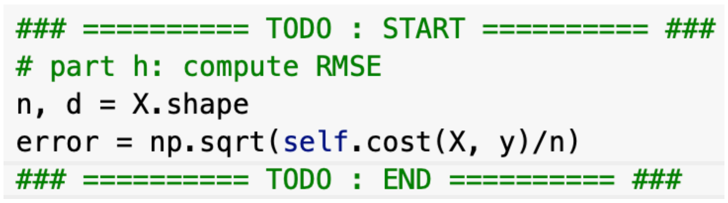


```
### ========== TODO : START ========== ###
# part b: modify to create matrix for simple linear model
# part g: modify to create matrix for polynomial model
# Phi = X
m = self.m_

Phi = np.empty((n,m+1))
for r in range(0, n):
  # first column of every row has value 1
  Phi[r][0] = 1

  # set the rest of the columns
  for c in range(1, m+1):
    Phi[r][c] = X[r][0]**c

### ========== TODO : END ========== ###
```

(h) Problem 4h

Solution:

Code for “PolynomialRegression.rms_error(..)”:



```
### ========== TODO : START ========== ###
# part h: compute RMSE
n, d = X.shape
error = np.sqrt(self.cost(X, y)/n)
### ========== TODO : END ========== ###
```

We prefer RMSE over $J(\theta)$ because for $J(\theta)$, larger data sets (ie data sets with larger values of n) will tend to have higher values for $J(\theta)$ because there are more points to contribute errors to the overall sum. Thus, a dataset with more data points will have a larger value of $J(\theta)$ than a dataset with less data points if both data sets have same average error per point.

We are probably more concerned about the average error per point because we want to know how well the model fits a dataset irrespective of dataset size. Therefore, we use RMSE because it adjusts for the differences in dataset sizes by dividing by the number of data points n.

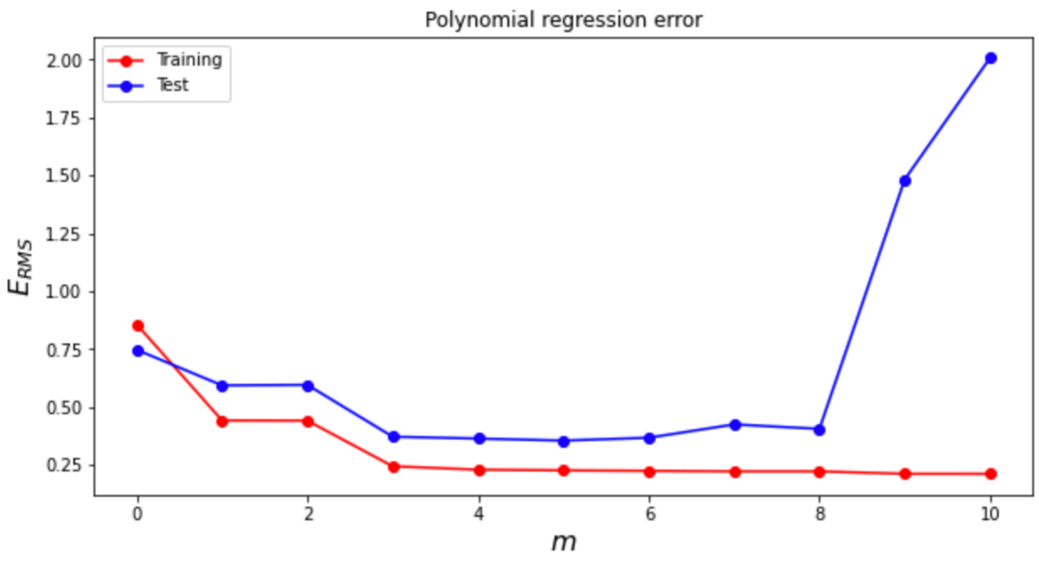
(i) Problem 4i

Solution:

Code in “main(..)” to generate the plot:

```
# part i: plot of train/test errors vs model complexity
print('********************** plot rms error **********************')
m = 10
train_error = []
test_error = []
for i in range(m+1):
  new_model = PolynomialRegression(i)
  new_model.fit(train_data.X, train_data.y)
  train_error.append(new_model.rms_error(train_data.X, train_data.y))
  test_error.append(new_model.rms_error(test_data.X, test_data.y))

print("train_error: ", train_error)
print("test_error: ", test_error)
plot_erms([i for i in range(m+1)], train_error, test_error)
```

Plot generated



I would say that the $m = 5$ degree polynomial fits the data best because $m = 5$ is the point on the x-axis at which the test error is lowest and the training error is relatively low.

There is underfitting on the left-side extreme (ie around $m < 3$) because both the training and test error are relatively high compared to their values around the middle (ie around $3 \leq m \leq 8$).

There is overfitting on the right-side extreme (ie around $m > 8$) because the training error is very low and the test error is very high, thus indicating that the data is closely fit to the training data and consequently fails to generalize well to the test set.