21W-COMSCIM146-1 Problem Set 2

ANH MAC

TOTAL POINTS

67 / 67

QUESTION 1

1 Perceptron 2 / 2

✓ **- 0 pts Correct**

- 0.25 pts Variables assumed to be in $$\{0,1\}$$

- 0.5 pts a) is incorrect but the right idea (e.g. the solution is correct for AND) or mistake in calculation

- 0.5 pts Only one solution in a) (including scaling of the parameters)

- 1 pts b) incorrect / no solution

- 1 pts No solution for a)

- 2 pts Late Submission

QUESTION 2

2 Logistic Regression 10 / 10

✓ **- 0 pts Correct**

- 2 pts (a) wrong

- 3 pts (b) wrong

- 5 pts (c) wrong

- 10 pts all wrong

- 1 pts trivial problem

- 2 pts trivial problem

- 3 pts trivial problem

- 10 pts Late Submssion

QUESTION 3

3 Maximum Likelihood Estimation 15 / 15

✓ **- 0 pts Correct**

- 1 pts Part (b): Negative second derivative and concavity proof.

- 1 pts Part (d): No (or imcomplete or incorrect) comparison on n=5 and n=100 (both $$\hat{\theta}_{MLE}=0.6$$)

- 1 pts Part (b): No explicit first derivative of $$l(\theta)$$

- 2 pts Part (b): No or wrong second derivative of $$l(\theta)$$

- 1 pts Part (b): Minor error in the second derivative

- 3 pts Part (c): No answer or wrong

- 3 pts Part (d): No answer or wrong

- 1 pts Part (a): An explicit and correct formula $$l(\theta)$$ of is expected.

- 1 pts Part (d): Missing Plot

- 1.5 pts Part (c): No explanation

- 2 pts Part (d): No explanation

- 1 pts Part (b): No explicit or incorrect $$\hat{\theta}_{MLE}$$

- 1 pts Part (c.d): Wrong plot

- 6 pts Part (b): Wrong

- 1 pts Part (a): Lack of explanation

- 2 pts Part (b): No explicit $$\hat{\theta}_{MLE}$$

- 2 pts Part (b): Unclear notation

- 15 pts Part (a,b,c): No answer

- 1 pts Part (a): Unclear/unrecognizable

QUESTION 4

Programming 40 pts

4.1 Visualization 2 / 2

✓ **- 0 pts Correct**

- 1 pts expected answer: linear regression will do poorly and data is non-linear

- 1 pts no plots

- 0.5 pts partially correct answer

4.2 Linear Regression: Part (b) 2 / 2

✓ **- 0 pts Correct**

- 2 pts no code/incomplete code/wrong code

- 1 pts column added after X

4.3 Linear Regression: Part (c) 3 / 3

✓ **- 0 pts Correct**

- **3 pts** no code/wrong code

4.4 Linear Regression: Part (d) 10 / 10

✓ - **0 pts** **Correct**

- **2 pts** cost is wrong
- **1 pts** cost code missing
- **2 pts** update code missing
- **4 pts** table wrong/not there
- **3 pts** update code wrong
- **2 pts** no table but some discussion / missing info in table

4.5 Linear Regression: Part (e) 4 / 4

✓ - **0 pts** **Correct**

- **2 pts** code missing
- **0.5 pts** time not discussed
- **0.5 pts** final cost not mentioned
- **4 pts** no answer/wrong answer

4.6 Linear Regression: Part (f) 4 / 4

✓ - **0 pts** **Correct**

- **1 pts** number of iterations not mentioned
- **2 pts** no code
- **2 pts** wrong code
- **4 pts** no answer
- **2 pts** num iters not given

4.7 Polynomial Regression: Part (g) 4 / 4

✓ - **0 pts** **Correct**

- **4 pts** no code

4.8 Polynomial Regression: Part (h) 4 / 4

✓ - **0 pts** **Correct**

- **2 pts** wrong code/no code
- **2 pts** expected: normalizes by the number of instances and/or represents the sample standard deviation of the
residuals and is therefore on the same scale as the predictions (compared to the the MSE,
which represents the sample variance).
- **1 pts** partially correct explanation

4.9 Polynomial Regression: Part (i) 7 / 7

✓ - **0 pts** **Correct**

- **1.5 pts** underfitting for m < 3 not mentioned
- **1.5 pts** overfitting for m > 8 not mentioned
- **1.5 pts** best degree not mentioned
- **7 pts** no answer
- **2 pts** plot looks incorrect
- **7 pts** no answer

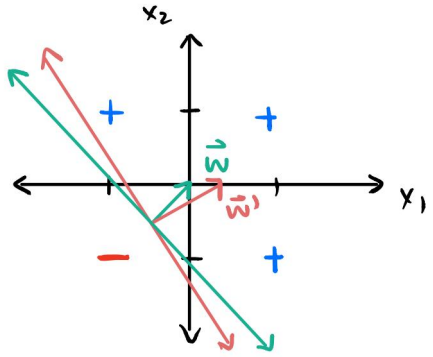
Anh Mac
905-111-606
CSM146, Winter 2021
Prof. Sankararaman

Problem Set 2: Perceptron and Regression

1. PERCEPTRON

a) OR

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



$$\vec{\theta} = \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} \qquad \vec{\theta}' = \begin{pmatrix} 2 \\ 2 \\ 3 \end{pmatrix}$$

$+1 - 1 + 1 = 1$
$+1 - 1 - 1 = -1$
$+1 + 1 + 1 = 3$
$+1 - 1 + 1 = 1$ ✓

$2 - 2 + 3 = 3$
$2 - 2 - 3 = -3$
$2 + 2 + 3 = 7$
$2 + 2 - 3 = 1$ ✓

$q = w^T x + b$
$\theta = (\theta_0, \theta_1, \theta_2)$
$x = (1, x_1, x_2)$
$y = \text{sign}(\theta^T x)$

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

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

$\theta_0 = \theta_1 = 1$

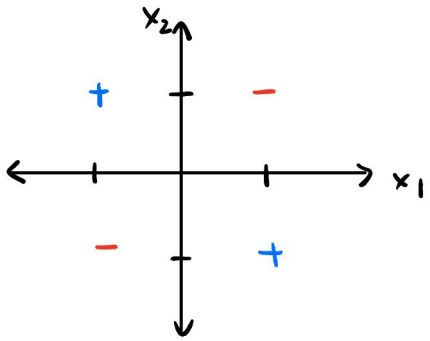
$-\theta_2 = -$
$+\theta_2 = +$

$\Rightarrow \theta_2 = 1$

$(1, 1, 1)$

b) XOR

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



No perceptron exists because the data is not linear separable.

1 Perceptron 2 / 2

✓ **- 0 pts Correct**

- 0.25 pts Variables assumed to be in $$\{0,1\}$$

- 0.5 pts a) is incorrect but the right idea (e.g. the solution is correct for AND) or mistake in calculation

- 0.5 pts Only one solution in a) (including scaling of the parameters)

- 1 pts b) incorrect / no solution

- 1 pts No solution for a)

- 2 pts Late Submission

2. LOGISTIC REGRESSION

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

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

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

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

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

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

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

$$= h_\theta(x)(1-h_\theta(x))\, x_k$$

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

$$= \frac{1}{h_\theta(x_n)}\, h_\theta(x_n)(1-h_\theta(x_n))\, x_j$$

$$= (1-h_\theta(x_n))\, x_j$$

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

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

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

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

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

$$= \boxed{\sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))\, x_{n,j}\, x_{n,k}}$$

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

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

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

$\therefore$ H is the Hessian. ■

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

$$= \sum_{n=1}^{N} h_\theta(x_n)(1-h_\theta(x_n))\, x_n x_n^T z^T z \qquad X = xx^T \text{ iff } x_{i,j} = x_i x_j$$

$$= \sum_{n=1}^{N} \underbrace{h_\theta(x_n)}_{[0,1]} \underbrace{(1-h_\theta(x_n))}_{[0,1)} \underbrace{(z^T x_n)^2}_{\geq 0}$$

We have following constraints: $0 \leq h_\theta(x_n) \leq 1$ and $(z^T x_n)^2 \geq 0$

so our expression

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

$\therefore H \succeq 0$ ■

2 Logistic Regression 10 / 10

✓ - **0 pts** **Correct**

- **2 pts** (a) wrong
- **3 pts** (b) wrong
- **5 pts** (c) wrong
- **10 pts** all wrong
- **1 pts** trivial problem
- **2 pts** trivial problem
- **3 pts** trivial problem
- **10 pts** Late Submssion

3. MAXIMUM LIKELIHOOD ESTIMATION

a) Given these conditions,

$$p(x_i) = \begin{cases} \theta & \text{if } x_i = 1 \\ 1-\theta & \text{if } x_i = 0 \end{cases}$$

we can derive the following formula:

$$p(x_i;\theta) = \theta^{x_i}(1-\theta)^{(1-x_i)}$$

The likelihood function becomes: $\boxed{L(\theta) = \prod_{i=1}^{n} p(x_i;\theta) = \prod_{i=1}^{n} \theta^{x_i}(1-\theta)^{(1-x_i)}}$

The likelihood function does not depend on the order in which the random variables are observed since we are given that the values x_i are independent random variables drawn from the same Bernoulli distribution with parameter θ, so they are IID. The likelihood is simply the product of all independent probability.

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

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

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

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

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

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

Since $\theta \in [0,1]$ and $x_i \in [0,1]$, the second derivative is negative (≤ 0) for all values of θ Thus, our log-likelihood function concaves down and the critical point would be the maximum, which is the MLE.

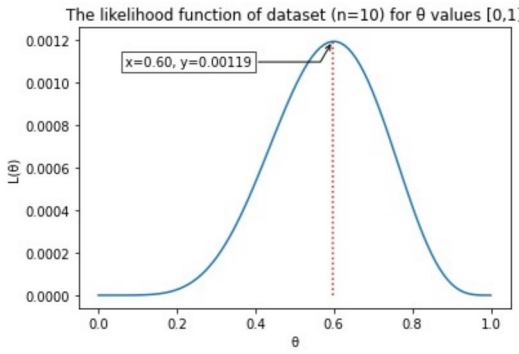
$$\ell'(\theta) = \sum_n \left[\frac{x_i}{\theta} - \frac{(1-x_i)}{1-\theta}\right] = 0$$

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

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

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

c)



n=10, dataset contains 6 1's and 4 0's

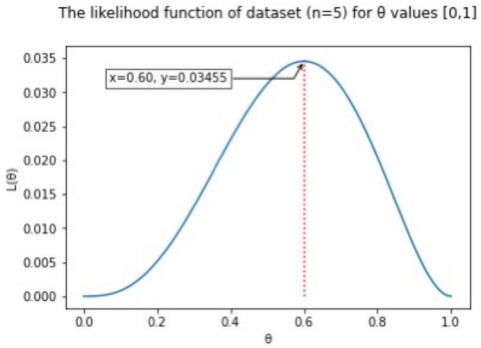
$\hat{\theta}_{MLE}$ is estimated to be at roughly $\theta = 0.6$.

Using the closed-form function for n=10,

we get $\theta = \frac{1}{10}\sum_{i=1}^{10} x_i = \frac{1(6)+0(4)}{10} = \frac{6}{10} = 0.6$

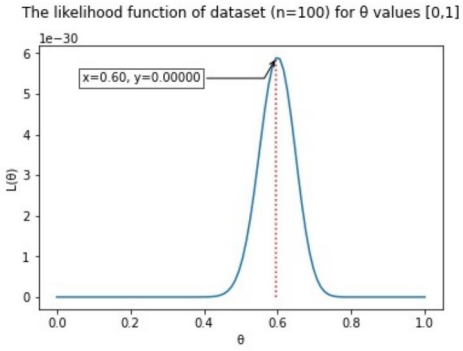
The answer agrees with the closed form answer.

d)



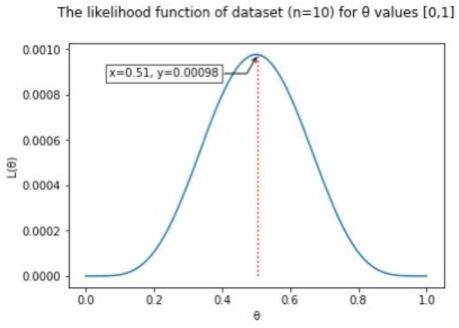
n=5, dataset contains 3 1's and 2 0's.

With a smaller data set with the same sample mean of 0.6, the likelihood function shows more variance (the graph appears wider). The $\hat{\theta}_{MLE}$ is still at roughly $\theta = 0.6$, but the likelihood $L(\theta)$ is at a greater maximum at $L(\theta) = 0.035$ compared to $L(\theta) = 0.00119$ previously for a bigger n=10 data set.



n=100, dataset contains 60 1's and 40 0's.

With a larger dataset with the same sample mean of 0.6, the likelihood function shows less variance (narrower curve). The $\hat{\theta}_{MLE}$ still occurs at $\theta = 0.60$, but the maximum likelihood $L(\theta)$ is significantly smaller than that of smaller data sets.



n=10, dataset contains 5 1's and 5 0's.

When the dataset has a different sample mean, $\hat{\theta}_{MLE}$ is now at $\theta \approx 0.51$, which still agrees with our closed form answer where

$$\theta = \frac{1(5)+0(5)}{10} = 0.50$$

The likelihood function shows similar variance to similar size n=10 data set. The maximum likelihood estimates is also of similar range, only slightly smaller (0.00098 vs. 0.00119).

3 Maximum Likelihood Estimation 15 / 15

✓ **- 0 pts Correct**

- 1 pts Part (b): Negative second derivative and concavity proof.

- 1 pts Part (d): No (or imcomplete or incorrect) comparison on n=5 and n=100 (both $$\hat{\theta}_{MLE}=0.6$$)

- 1 pts Part (b): No explicit first derivative of $$l(\theta)$$

- 2 pts Part (b): No or wrong second derivative of $$l(\theta)$$

- 1 pts Part (b): Minor error in the second derivative

- 3 pts Part (c): No answer or wrong

- 3 pts Part (d): No answer or wrong

- 1 pts Part (a): An explicit and correct formula $$l(\theta)$$ of is expected.

- 1 pts Part (d): Missing Plot

- 1.5 pts Part (c): No explanation

- 2 pts Part (d): No explanation

- 1 pts Part (b): No explicit or incorrect $$\hat{\theta}_{MLE}$$

- 1 pts Part (c.d): Wrong plot

- 6 pts Part (b): Wrong

- 1 pts Part (a): Lack of explanation

- 2 pts Part (b): No explicit $$\hat{\theta}_{MLE}$$

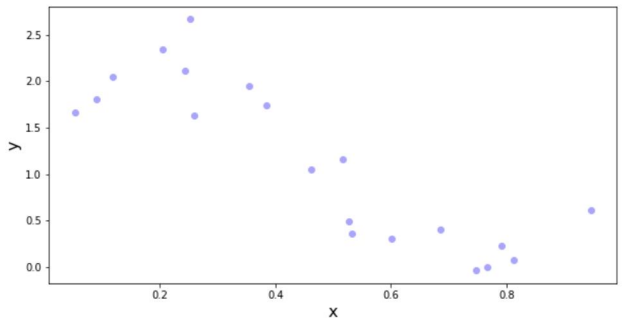
- 2 pts Part (b): Unclear notation

- 15 pts Part (a,b,c): No answer

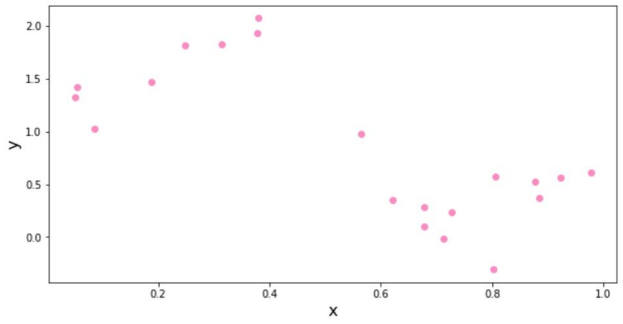
- 1 pts Part (a): Unclear/unrecognizable

4. a)

```
Visualizing data...
------TRAINING DATA------
```




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



Both of the datasets don't show a strong linear relationship between the data points. Because of this, linear regression may not be effective and will predict poorly given this dataset.

The data resembles more of a polynomial relationship, so a polynomial regression would be a more effective predictor.

```
### ========== TODO : START ========== ###
# part a: main code for visualizations
print('Visualizing data...')
print("------TRAINING DATA------")
plot_data(train_data.X, train_data.y, color='#aba9fc')
print("------TEST DATA------")
plot_data(test_data.X, test_data.y, color='#fa8ec6')
### ========== TODO : END ========== ###
```

b)

```
def generate_polynomial_features(self, X) :
    """
    Maps X to an mth degree feature vector e.g. [1, X, X^2, ..., X^m].

    Parameters
    --------------------
        X       -- numpy array of shape (n,1), features

    Returns
    --------------------
        Phi     -- numpy array of shape (n,(m+1)), mapped features
    """

    n,d = X.shape

    ### ========== TODO : START ========== ###
    # part b: modify to create matrix for simple linear model
    # part g: modify to create matrix for polynomial model
    m = self.m_
    X0 = np.ones((n,1)) # create column of 1's to add additional "feature" to each instance
    Xnew = np.hstack((X0,X)) # append as first column to X
    Phi = Xnew

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

    return Phi
```

c)

```
def predict(self, X) :
    """
    Predict output for X.

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

    Returns
    --------------------
        y       -- numpy array of shape (n,), predictions
    """
    if self.coef_ is None :
        raise Exception("Model not initialized. Perform a fit first.")

    X = self.generate_polynomial_features(X) # map features

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

    return y
```

4.1 Visualization 2 / 2

✓ **- 0 pts Correct**

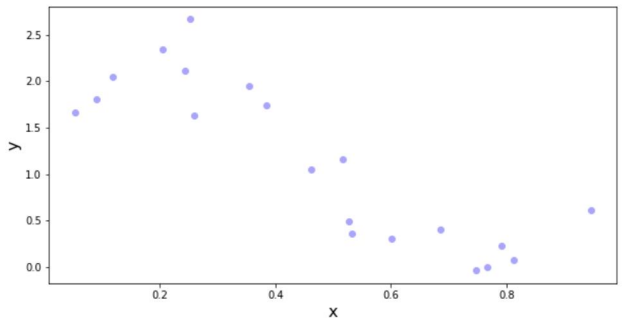
- 1 pts expected answer: linear regression will do poorly and data is non-linear

- 1 pts no plots

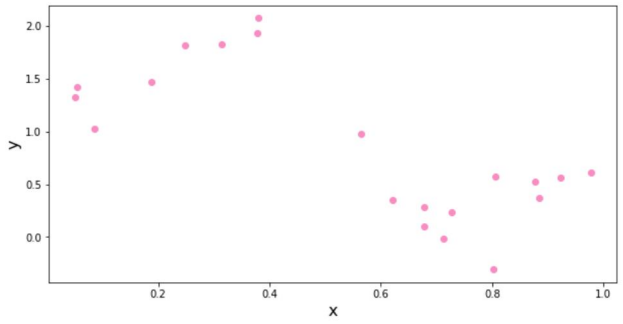
- 0.5 pts partially correct answer

4. a)

```
Visualizing data...
------TRAINING DATA------
```




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



Both of the datasets don't show a strong linear relationship between the data points. Because of this, linear regression may not be effective and will predict poorly given this dataset.

The data resembles more of a polynomial relationship, so a polynomial regression would be a more effective predictor.

```
### ========== TODO : START ========== ###
# part a: main code for visualizations
print('Visualizing data...')
print("------TRAINING DATA------")
plot_data(train_data.X, train_data.y, color='#aba9fc')
print("------TEST DATA------")
plot_data(test_data.X, test_data.y, color='#fa8ec6')
### ========== TODO : END ========== ###
```

b)

```
def generate_polynomial_features(self, X) :
    """
    Maps X to an mth degree feature vector e.g. [1, X, X^2, ..., X^m].

    Parameters
    --------------------
        X       -- numpy array of shape (n,1), features

    Returns
    --------------------
        Phi     -- numpy array of shape (n,(m+1)), mapped features
    """

    n,d = X.shape

    ### ========== TODO : START ========== ###
    # part b: modify to create matrix for simple linear model
    # part g: modify to create matrix for polynomial model
    m = self.m_
    X0 = np.ones((n,1)) # create column of 1's to add additional "feature" to each instance
    Xnew = np.hstack((X0,X)) # append as first column to X
    Phi = Xnew

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

    return Phi
```

c)

```
def predict(self, X) :
    """
    Predict output for X.

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

    Returns
    --------------------
        y       -- numpy array of shape (n,), predictions
    """
    if self.coef_ is None :
        raise Exception("Model not initialized. Perform a fit first.")

    X = self.generate_polynomial_features(X) # map features

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

    return y
```

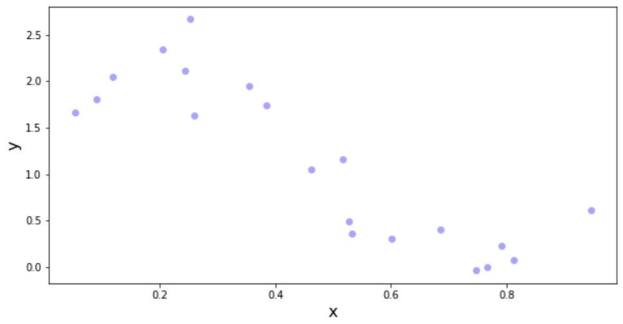
4.2 Linear Regression: Part (b) 2 / 2

✓ - **0 pts** **Correct**

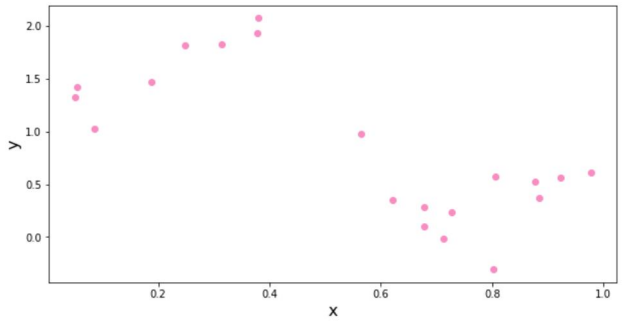
- **2 pts** no code/incomplete code/wrong code
- **1 pts** column added after X

4. a)

```
Visualizing data...
------TRAINING DATA------
```




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



Both of the datasets don't show a strong linear relationship between the data points. Because of this, linear regression may not be effective and will predict poorly given this dataset.

The data resembles more of a polynomial relationship, so a polynomial regression would be a more effective predictor.

```
### ========== TODO : START ========== ###
# part a: main code for visualizations
print('Visualizing data...')
print("------TRAINING DATA------")
plot_data(train_data.X, train_data.y, color='#aba9fc')
print("------TEST DATA------")
plot_data(test_data.X, test_data.y, color='#fa8ec6')
### ========== TODO : END ========== ###
```

b)

```
def generate_polynomial_features(self, X) :
    """
    Maps X to an mth degree feature vector e.g. [1, X, X^2, ..., X^m].

    Parameters
    --------------------
        X       -- numpy array of shape (n,1), features

    Returns
    --------------------
        Phi     -- numpy array of shape (n,(m+1)), mapped features
    """

    n,d = X.shape

    ### ========== TODO : START ========== ###
    # part b: modify to create matrix for simple linear model
    # part g: modify to create matrix for polynomial model
    m = self.m_
    X0 = np.ones((n,1)) # create column of 1's to add additional "feature" to each instance
    Xnew = np.hstack((X0,X)) # append as first column to X
    Phi = Xnew

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

    return Phi
```

c)

```
def predict(self, X) :
    """
    Predict output for X.

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

    Returns
    --------------------
        y       -- numpy array of shape (n,), predictions
    """
    if self.coef_ is None :
        raise Exception("Model not initialized. Perform a fit first.")

    X = self.generate_polynomial_features(X) # map features

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

    return y
```

4.3 Linear Regression: Part (c) 3 / 3

✓ **- 0 pts** **Correct**

- 3 pts no code/wrong code

d)

```
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 running the code from the spec using this implementation, it returned a value of 40.233847 which is 40.234, as expected.

```
### ========== 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)
xTx_theta = np.dot(xTx, self.coef_)
xTy = np.dot(X.T, y)
gradient = xTx_theta - xTy
self.coef_ = np.subtract(self.coef_, 2*eta*gradient) # update theta

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

Learning Rate (η)	Coefficients	Number 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
0.0168	[2.44640706; -2.81635353]	470	3.91257640579147

The algorithm did not converge when running with $\eta = 10^{-6}$ and 10^{-5} because the number of iterations reaches 10,000. For the first 2 rows, the algorithm was not close to convergence since the final cost value was still quite large. However, for the algorithm running with $\eta = 10^{-3}$, the algorithm converged with 7,055 iterations.

When $\eta = 0.0168$, the algorithm only took 470 iterations to converge, and its final coefficients and final cost value are very similar to that of $\eta = 10^{-3}$. The only difference is that it took less iterations than what it took when η was a smaller value at 10^{-3} to converge and achieve the same result.

This indicates that it is important to choose an appropriate step size for the gradient descent algorithm to run efficiently. Too large or too small may cause the algorithm to take longer to converge or even not converging at all.

4.4 Linear Regression: Part (d) 10 / 10

✓ - 0 pts Correct

- 2 pts cost is wrong
- 1 pts cost code missing
- 2 pts update code missing
- 4 pts table wrong/not there
- 3 pts update code wrong
- 2 pts no table but some discussion / missing info in table

e)

```
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
    xTx = np.dot(X.T,X)
    xTx1 = np.linalg.pinv(np.dot(X.T,X))
    xTy = np.dot(X.T,y)
    self.coef_ = np.dot(xTx1, xTy)
    ### ========== TODO : END ========== ###
```

The closed form solution coefficients and cost are:

Coefficients	[2.44640709; -2.81635359]
Cost	3.9125764057914636

Both values are the similar with the results obtained with GD when using larger step sizes (10^{-3} and 0.0168). The closed form algorithm runs faster than GD since it only has to compute the matrix computations once while GD has to perform such computations over an amount of iterations until convergence.

f)

```
### ========== 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) # change this line
else :
    eta = eta_input
### ========== TODO : END ========== ###
```

When setting the learning rate as a function of k where $\eta_{1k} = \frac{1}{1+k}$, it took the algorithm 1350 iterations to converge to coefficients of [2.4464, -2.8164]. The final cost and coefficients are as follows:

Coefficients	[2.44640744; -2.81635429]
Cost	3.912576405792132

g)

```
def generate_polynomial_features(self, X) :
    """
    Maps X to an mth degree feature vector e.g. [1, X, X^2, ..., X^m].

    Parameters
    --------------------
        X       -- numpy array of shape (n,1), features

    Returns
    --------------------
        Phi     -- numpy array of shape (n,(m+1)), mapped features
    """

    n,d = X.shape

    ### ========== TODO : START ========== ###
    # part b: modify to create matrix for simple linear model

    # part g: modify to create matrix for polynomial model
    m = self.m_
    polynomial_basis = [i for i in range(0,m+1)] # create a m+1-dimensional feature vector [1, x, x^2, ... , x^m] for each instance
    Xnew = np.power(X, polynomial_basis)

    Phi = Xnew
    ### ========== TODO : END ========== ###

    return Phi
```

4.5 Linear Regression: Part (e) 4 / 4

✓ **- 0 pts Correct**

- 2 pts code missing

- 0.5 pts time not discussed

- 0.5 pts final cost not mentioned

- 4 pts no answer/wrong answer

e)

```
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
    xTx = np.dot(X.T,X)
    xTx1 = np.linalg.pinv(np.dot(X.T,X))
    xTy = np.dot(X.T,y)
    self.coef_ = np.dot(xTx1, xTy)
    ### ========== TODO : END ========== ###
```

The closed form solution coefficients and cost are:

Coefficients	[2.44640709; -2.81635359]
Cost	3.9125764057914636

Both values are the similar with the results obtained with GD when using larger step sizes (10^{-3} and 0.0168). The closed form algorithm runs faster than GD since it only has to compute the matrix computations once while GD has to perform such computations over an amount of iterations until convergence.

f)

```
### ========== 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) # change this line
else :
    eta = eta_input
### ========== TODO : END ========== ###
```

When setting the learning rate as a function of k where $\eta_{1k} = \frac{1}{1+k}$, it took the algorithm 1350 iterations to converge to coefficients of [2.4464, -2.8164]. The final cost and coefficients are as follows:

Coefficients	[2.44640744; -2.81635429]
Cost	3.912576405792132

g)

```
def generate_polynomial_features(self, X) :
    """
    Maps X to an mth degree feature vector e.g. [1, X, X^2, ..., X^m].

    Parameters
    --------------------
        X       -- numpy array of shape (n,1), features

    Returns
    --------------------
        Phi     -- numpy array of shape (n,(m+1)), mapped features
    """

    n,d = X.shape

    ### ========== TODO : START ========== ###
    # part b: modify to create matrix for simple linear model

    # part g: modify to create matrix for polynomial model
    m = self.m_
    polynomial_basis = [i for i in range(0,m+1)] # create a m+1-dimensional feature vector [1, x, x^2, ... , x^m] for each instance
    Xnew = np.power(X, polynomial_basis)

    Phi = Xnew
    ### ========== TODO : END ========== ###

    return Phi
```

4.6 Linear Regression: Part (f) 4 / 4

✓ **- 0 pts Correct**

- 1 pts number of iterations not mentioned

- 2 pts no code

- 2 pts wrong code

- 4 pts no answer

- 2 pts num iters not given

e)

```
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
    xTx = np.dot(X.T,X)
    xTx1 = np.linalg.pinv(np.dot(X.T,X))
    xTy = np.dot(X.T,y)
    self.coef_ = np.dot(xTx1, xTy)
    ### ========== TODO : END ========== ###
```

The closed form solution coefficients and cost are:

Coefficients	[2.44640709; -2.81635359]
Cost	3.9125764057914636

Both values are the similar with the results obtained with GD when using larger step sizes (10^{-3} and 0.0168). The closed form algorithm runs faster than GD since it only has to compute the matrix computations once while GD has to perform such computations over an amount of iterations until convergence.

f)

```
### ========== 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) # change this line
else :
    eta = eta_input
### ========== TODO : END ========== ###
```

When setting the learning rate as a function of k where $\eta_{1k} = \frac{1}{1+k}$, it took the algorithm 1350 iterations to converge to coefficients of [2.4464, -2.8164]. The final cost and coefficients are as follows:

Coefficients	[2.44640744; -2.81635429]
Cost	3.912576405792132

g)

```
def generate_polynomial_features(self, X) :
    """
    Maps X to an mth degree feature vector e.g. [1, X, X^2, ..., X^m].

    Parameters
    --------------------
        X       -- numpy array of shape (n,1), features

    Returns
    --------------------
        Phi     -- numpy array of shape (n,(m+1)), mapped features
    """

    n,d = X.shape

    ### ========== TODO : START ========== ###
    # part b: modify to create matrix for simple linear model

    # part g: modify to create matrix for polynomial model
    m = self.m_
    polynomial_basis = [i for i in range(0,m+1)] # create a m+1-dimensional feature vector [1, x, x^2, ... , x^m] for each instance
    Xnew = np.power(X, polynomial_basis)

    Phi = Xnew
    ### ========== TODO : END ========== ###

    return Phi
```

4.7 Polynomial Regression: Part (g) 4 / 4

✓ - 0 pts Correct

- 4 pts no code

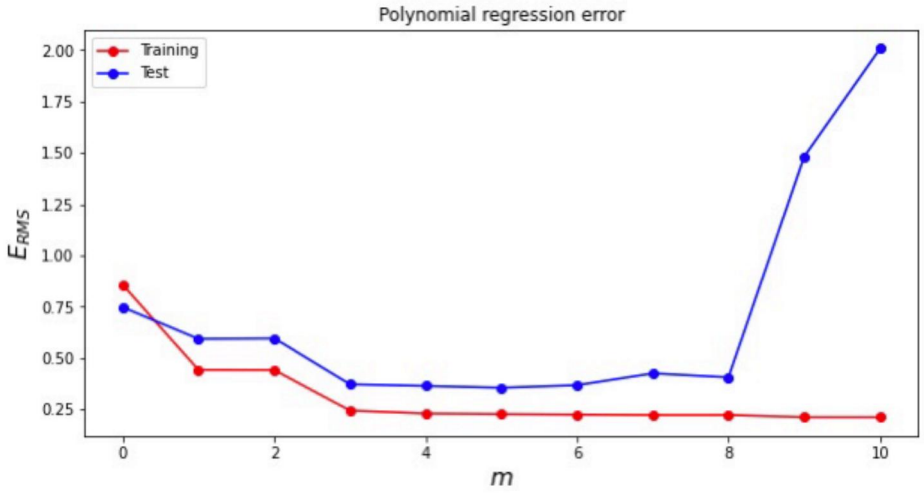
h) We prefer RMSE as a metric because since it considered the set size and normalize our objective function $J(\theta)$ using size of N instances, our result can be generalized and more comparable across different sized data sets. RMS also is representative of the sample standard deviation of the residuals, whereas MSE is representative of the sample variance. Because of this, we might prefer RMS over MSE as it would be on the same scale of our predictions.

```
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
```

i)



The degree polynomials that would best fit this dataset would be $m = 3,4,5,6$ because it is where both the training and test errors are the lowest. We can see overfitting occurring with $m > 8$ as the training error is low while the test error is high, which means our model did not generalize well. For $m < 3$, underfitting occurs as both the training and test error are high.

4.8 Polynomial Regression: Part (h) 4 / 4

✓ **- 0 pts Correct**

- 2 pts wrong code/no code

- 2 pts expected: normalizes by the number of instances and/or represents the sample standard deviation of the
residuals and is therefore on the same scale as the predictions (compared to the the MSE,
which represents the sample variance).

- 1 pts partially correct explanation

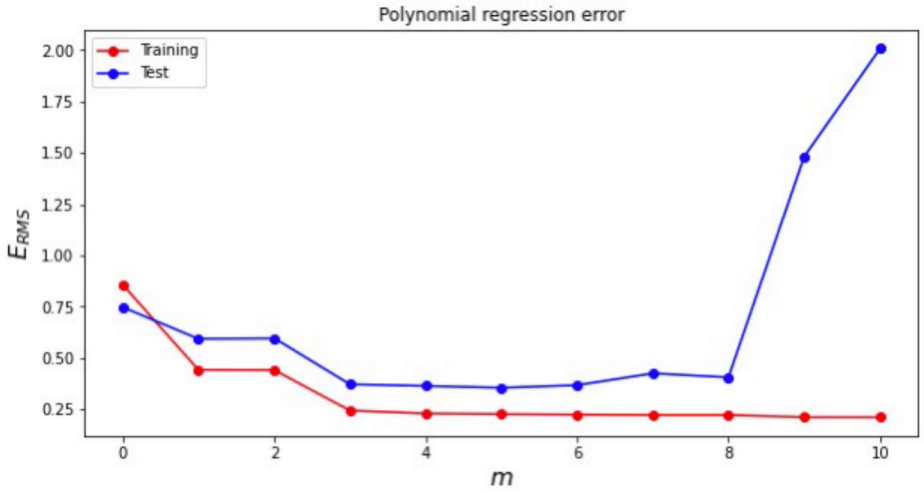
h) We prefer RMSE as a metric because since it considered the set size and normalize our objective function $J(\theta)$ using size of N instances, our result can be generalized and more comparable across different sized data sets. RMS also is representative of the sample standard deviation of the residuals, whereas MSE is representative of the sample variance. Because of this, we might prefer RMS over MSE as it would be on the same scale of our predictions.

```
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
```

i)



The degree polynomials that would best fit this dataset would be $m = 3,4,5,6$ because it is where both the training and test errors are the lowest. We can see overfitting occurring with $m > 8$ as the training error is low while the test error is high, which means our model did not generalize well. For $m < 3$, underfitting occurs as both the training and test error are high.

4.9 Polynomial Regression: Part (i) 7 / 7

✓ **- 0 pts Correct**

- 1.5 pts underfitting for m < 3 not mentioned

- 1.5 pts overfitting for m > 8 not mentioned

- 1.5 pts best degree not mentioned

- 7 pts no answer

- 2 pts plot looks incorrect

- 7 pts no answer