1 Perceptron [2 pts]

Design (specify θ for) a two-input perceptron (with an additional bias or offset term) that computes the following boolean functions. Assume T = 1 and F = −1. If a valid perceptron exists, show that it is not unique by designing another valid perceptron (with a different hyperplane, not simply through normalization). If no perceptron exists, state why.

(a) OR

a) OR

X1	X2	y
F	F	F
F	T	T
T	F	T
T	T	T

$$\theta_1(x_1,x_2) = \begin{bmatrix}1 & 1\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} + 1$$

	out
$\theta(-1,-1) = -1$	-1
$\theta(-1,1) = 1$	1
$\theta(1,-1) = 1$	1
$\theta_1(1,1) = 3$	1

$\boxed{\theta_1 = (1, 1, 1)}$

Not unique: another perceptron

$$\theta_2(x_1,x_2) = \begin{bmatrix}1 & 2\end{bmatrix}\begin{bmatrix}x_1\\ x_2\end{bmatrix} + 2$$

$\boxed{\theta_2 = (1, 2, 2)}$

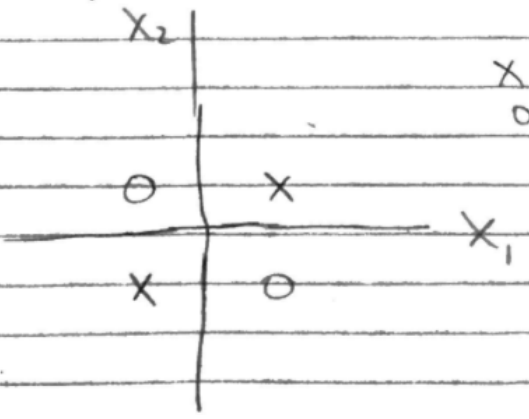
	out
$\theta_2(-1,-1) = -1$	-1
$\theta_2(-1,1) = 3$	1
$\theta_3(1,-1) = 1$	1
$\theta_4(1,1) = 5$	1

(b) XOR

b) XOR

X_1	X_2	y
F	F	F
F	T	T
T	F	T
T	T	F

No valid perceptron exists for XOR because it is not linearly separable



x is -1 (F)

o is 1 (T)

Not linearly separable

2 Logistic Regression [10 pts]

Consider the objective function that we minimize in logistic regression:

$$J(\boldsymbol{\theta}) = -\sum_{n=1}^{N}\left[y_n \log h_{\boldsymbol{\theta}}(\boldsymbol{x}_n) + (1-y_n)\log\left(1-h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\right)\right]$$

(a) Find the partial derivatives $\frac{\partial J}{\partial \theta_j}$.

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

a) $\frac{\partial J}{\partial \theta_j} = ?$ $\qquad 1-h_\theta(x_n) = \frac{1+e^{-\theta^T x_n}}{1+e^{-\theta^T x_n}} - \frac{1}{1+e^{-\theta^T x_n}}$

$$= \frac{-\partial}{\partial \theta_j}\sum_{n=1}^{N}\left[y_n \log\frac{1}{1+e^{-\theta^T x_n}} + (1-y_n)\log\frac{e^{-\theta^T x_n}}{1+e^{-\theta^T x_n}}\right]$$

$$= \frac{-\partial}{\partial \theta_j}\sum_{n=1}^{N}\left[y_n' \log(1+e^{-\theta^T x_n}) + (1-y_n)\left(\log e^{-\theta^T x_n} - \log(1+e^{-\theta^T x_n})\right)\right]$$

$$= \frac{-\partial}{\partial \theta_j}\sum_{n=1}^{N}\left[-y_n \log(1+e^{-\theta^T x_n}) + (1-y_n)\left(-\theta^T x_n - \log(1+e^{-\theta^T x_n})\right)\right]$$

$$\frac{\partial}{\partial \theta_j}\left(\log(1+e^{-\theta^T x_n})\right) = \frac{1}{1+e^{-\theta^T x_n}} \cdot \frac{\partial}{\partial \theta_j}\left(1+e^{-\theta^T x_n}\right)$$

$$= \frac{1}{1+e^{-\theta^T x_n}} \cdot \left(-x_{nj}\, e^{-\theta^T x_n}\right)$$

$$= -\sum_{n=1}^{N}\left[y_n x_{nj}\frac{e^{-\theta^T x_n}}{1+e^{-\theta^T x_n}} + (1-y_n)\left(-x_{nj} + x_{nj}\frac{e^{-\theta^T x_n}}{1+e^{-\theta^T x_n}}\right)\right]$$

$$= -\sum_{n=1}^{N} x_{nj}\left[y_n \frac{e^{-\theta^T x_n}}{1+e^{-\theta^T x_n}} + (1-y_n)\left(\frac{e^{-\theta^T x_n}}{1+e^{-\theta^T x_n}} - 1\right)\right]$$

$$= -\sum_{n=1}^{N} x_{nj}\left[y_n \frac{e^{-\theta^T x_n}}{1+e^{-\theta^T x_n}} + \frac{e^{-\theta^T x_n}}{1+e^{-\theta^T x_n}} - 1 - y_n\frac{e^{-\theta^T x_n}}{1+e^{-\theta^T x_n}} + y_n\right]$$

$$= -\sum_{n=1}^{N} x_{nj}\left[\frac{e^{-\theta^T x_n}}{1+e^{-\theta^T x_n}} + y_n - 1\right] = -\sum_{n=1}^{N} x_{nj}\left[y_n - \underbrace{\frac{1}{1+e^{-\theta^T x_n}}}_{h_\theta(x_n)}\right]$$

$$= \boxed{\sum_{n=1}^{N} x_{nj}\left(h_\theta(x_n) - y_n\right)}$$

Note, for the bias term, $x_{nj} = 1$

(b) Find the partial second derivatives $\frac{\partial^2 J}{\partial \theta_j \partial \theta_k}$ and show that the Hessian (the matrix $\boldsymbol{H}$ of second derivatives with elements $H_{jk} = \frac{\partial^2 J}{\partial \theta_j \partial \theta_k}$) can be written as $\boldsymbol{H} = \sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\left(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)\right)\boldsymbol{x}_n\boldsymbol{x}_n^T$.

b) $$\frac{\partial J}{\partial \theta_j} = \sum_{n=1} x_{nj}\left(h_\theta(x_n) - y_n\right)$$

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

$$= \sum_{n=1}^{N}\left(x_{nj}\frac{\partial}{\partial \theta_k}\left(h_\theta(x_n)\right) - \frac{\partial}{\partial \theta_k}\left(x_n y_n\right)\right)$$

$$= \sum_{n=1}^{N} x_{nj}\frac{\partial}{\partial \theta_k}\left(\frac{1}{1+e^{-\theta^T x_n}}\right)$$

$$\frac{\partial}{\partial \theta_k}(1+e^{-\theta^T x_n})^{-1} = -(1+e^{-\theta^T x_n})^{-2}(-x_{nk}e^{-\theta^T x_n})$$

$$= \frac{x_{nk}e^{-\theta^T x_n}}{(1+e^{-\theta^T x_n})^2} = x_{nk}\left(\frac{1}{1+e^{-\theta^T x_n}}\right)\left(\frac{e^{-\theta^T x_n}}{1+e^{-\theta^T x_n}}\right)$$

$$= x_{nk} \cdot h_\theta(x_n)(1-h_\theta(x_n))$$

$$H_{jk} = \sum_{n=1}^{N} x_{nj}\, x_{nk}\, h_\theta(x_n)(1-h_\theta(x_n))$$

Entry j,k in H

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

$x_n x_n^T$ forms matrix, each entry is $x_{nj} \cdot x_{nk}$

(c) Show that J is a convex function and therefore has no local minima other than the global one.
Hint: A function J is convex if its Hessian is positive semi-definite (PSD), written $\boldsymbol{H} \succeq 0$. A matrix is PSD if and only if

$$\boldsymbol{z}^T \boldsymbol{H} \boldsymbol{z} \equiv \sum_{j,k} z_j z_k H_{jk} \geq 0.$$

for all real vectors $\boldsymbol{z}$.

c) J is convex if $H \succeq 0$, Also,

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

$$z_j z_k H_{jk} = \sum_{n=1}^{N} a_{nj} a_{nk} x_{nj} x_{nk} h_\theta(x_n)(1-h_\theta(x_n))$$

$$z^T H z = \sum_{n=1}^{N} \underbrace{h_\theta(x_n)(1-h_\theta(x_n))}_{\text{positive: product of two positives } (h_\theta(x_n) \text{ is bound by 0 and 1})} \underbrace{x_n x_n^T}_{\text{squaring dot product}} \underbrace{a_n a_n^T}_{\text{Squaring dot product}}$$

$\boxed{\text{Sum over series of products of 3 positives is positive so } H \succeq 0}$

3 Maximum Likelihood Estimation [15 pts]

Suppose we observe the values of n independent random variables $X_1, \dots, X_n$ drawn from the same Bernoulli distribution with parameter θ[1]. In other words, for each X_i, we know that

$$P(X_i = 1) = \theta \quad \text{and} \quad P(X_i = 0) = 1 - \theta.$$

Our goal is to estimate the value of θ from these observed values of X_1 through X_n.

For any hypothetical value $\hat{\theta}$, we can compute the probability of observing the outcome $X_1, \dots, X_n$ if the true parameter value θ were equal to $\hat{\theta}$. This probability of the observed data is often called the *likelihood*, and the function $L(\theta)$ that maps each θ to the corresponding likelihood is called the *likelihood function*. A natural way to estimate the unknown parameter θ is to choose the θ that maximizes the likelihood function. Formally,

$$\hat{\theta}_{MLE} = \arg\max_{\theta} L(\theta).$$

(a) Write a formula for the likelihood function, $L(\theta) = P(X_1, \dots, X_n; \theta)$. Your function should depend on the random variables $X_1, \dots, X_n$ and the hypothetical parameter θ. Does the likelihood function depend on the order in which the random variables are observed ?

$P(X_i=1)=\theta \quad P(x_i=0)=1-\theta$

$\hat{\theta}_{MLE} = \arg_\theta \max L(\theta)$

a) $L(\theta) = P(X_1, \dots, X_n; \theta) = \prod_{i=1}^{n} P_\theta(X_i)$

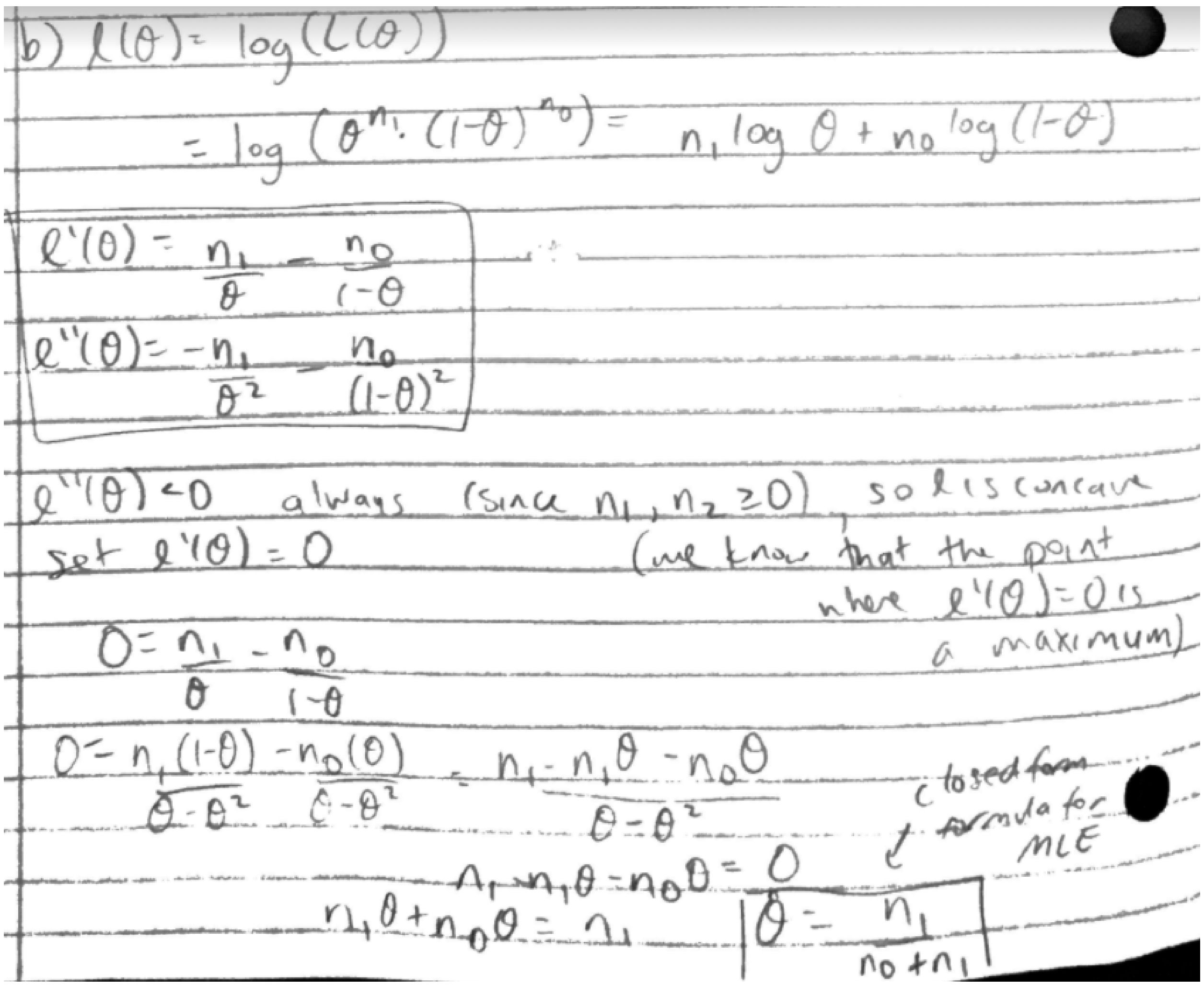
$= \boxed{\prod_{i=1}^{n} \left(\underset{X_i=1}{\theta^{X_i}} \cdot \underset{X_i=0}{(1-\theta)^{1-X_i}} \right)}$

The likelihood function does not depend on the order of the data

Let n_0 be the number of $X_i=0$ and n_1 be the number of $X_i=1$

$\boxed{L(\theta) = \theta^{n_1} \cdot (1-\theta)^{n_0}}$

(b) Since the log function is increasing, the θ that maximizes the *log likelihood* $\ell(\theta) = \log(L(\theta))$ is the same as the θ that maximizes the likelihood. Find $\ell(\theta)$ and its first and second derivatives, and use these to find a closed-form formula for the MLE.


b) $\ell(\theta) = \log(L(\theta))$

$= \log\left(\theta^{n_1} \cdot (1-\theta)^{n_0}\right) = n_1 \log \theta + n_0 \log(1-\theta)$

$$\boxed{\begin{aligned} \ell'(\theta) &= \frac{n_1}{\theta} - \frac{n_0}{1-\theta} \\ \ell''(\theta) &= -\frac{n_1}{\theta^2} - \frac{n_0}{(1-\theta)^2} \end{aligned}}$$

$\ell''(\theta) < 0$ always (since $n_1, n_2 \geq 0$), so ℓ is concave

set $\ell'(\theta) = 0$ (we know that the point where $\ell'(\theta) = 0$ is a maximum)

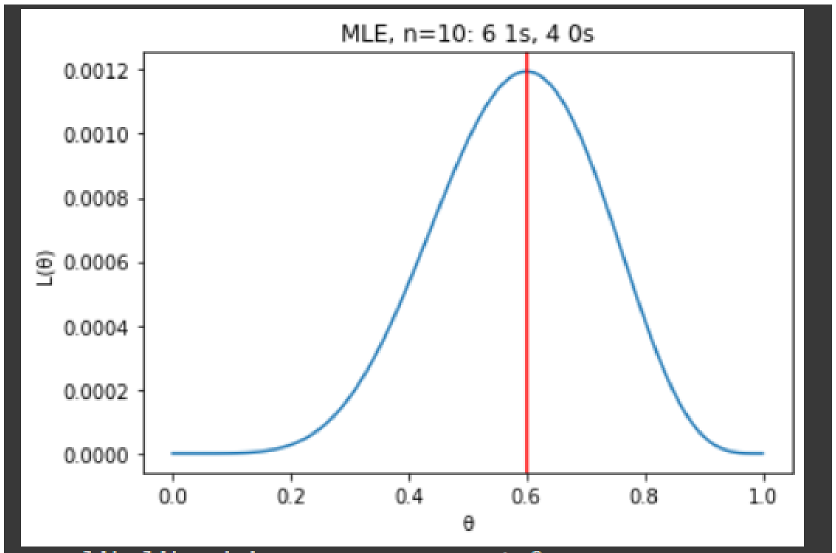
$0 = \frac{n_1}{\theta} - \frac{n_0}{1-\theta}$

$0 = \frac{n_1(1-\theta)}{\theta-\theta^2} - \frac{n_0(\theta)}{\theta-\theta^2} = \frac{n_1 - n_1\theta - n_0\theta}{\theta-\theta^2}$

$n_1 - n_1\theta - n_0\theta = 0$

$n_1\theta + n_0\theta = n_1$ $\quad \boxed{\theta = \frac{n_1}{n_0 + n_1}}$ closed form formula for MLE

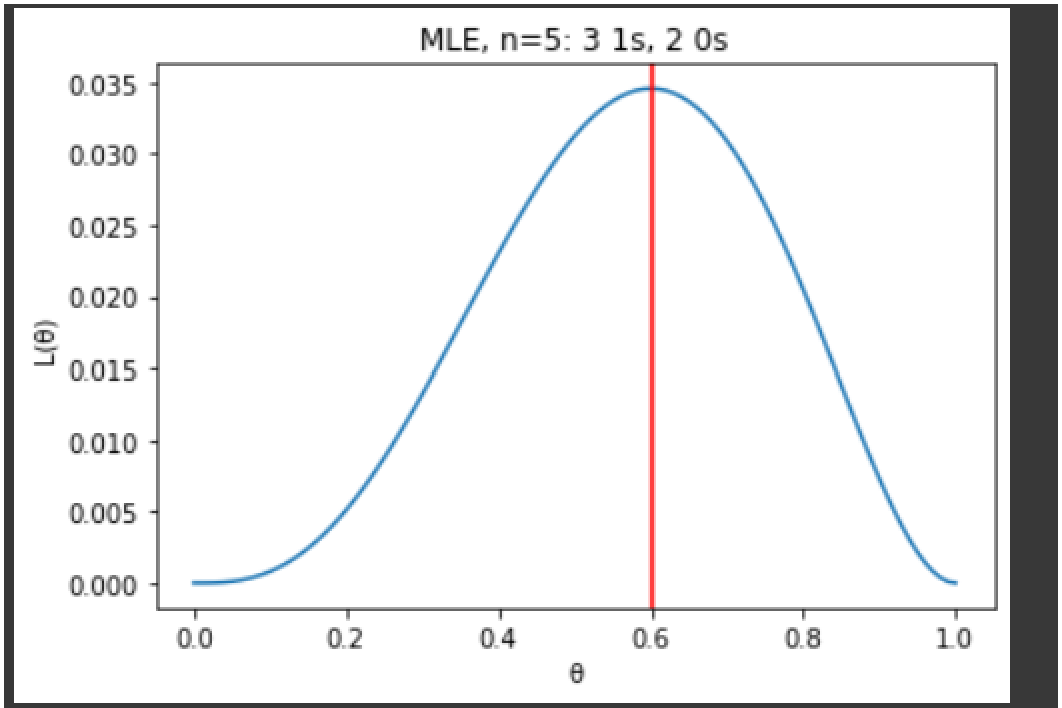
(c) Suppose that $n = 10$ and the data set contains six 1s and four 0s. Write a short program `likelihood.py` that plots the likelihood function of this data for each value of $\hat{\theta}$ in $\{0, 0.01, 0.02, \ldots, 1.0\}$ (use `np.linspace(...)` to generate this spacing). For the plot, the x-axis should be θ and the y-axis $L(\theta)$. Scale your y-axis so that you can see some variation in its value. Include the plot in your writeup (there is no need to submit your code). Estimate $\hat{\theta}_{MLE}$ by marking on the x-axis the value of $\hat{\theta}$ that maximizes the likelihood. Does the answer agree with the closed form answer ?



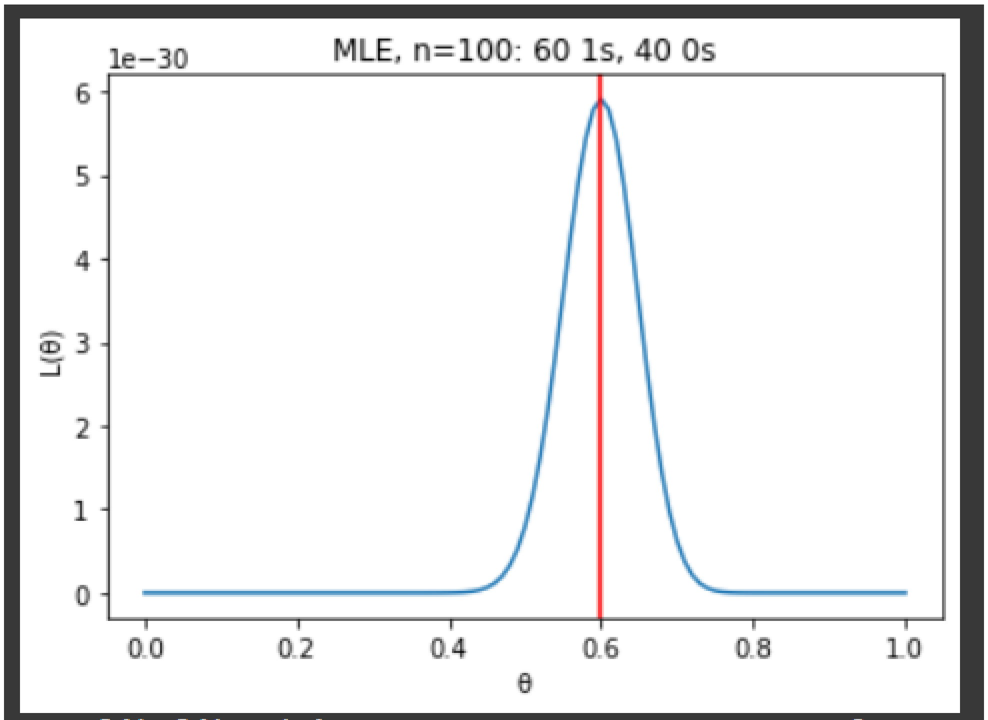
```
Max likelihood is 0.0011943936 at θ=0.6
```

This answer agrees with the closed form answer: θ = n1 / (n0+n1) = 6/10 = 0.6

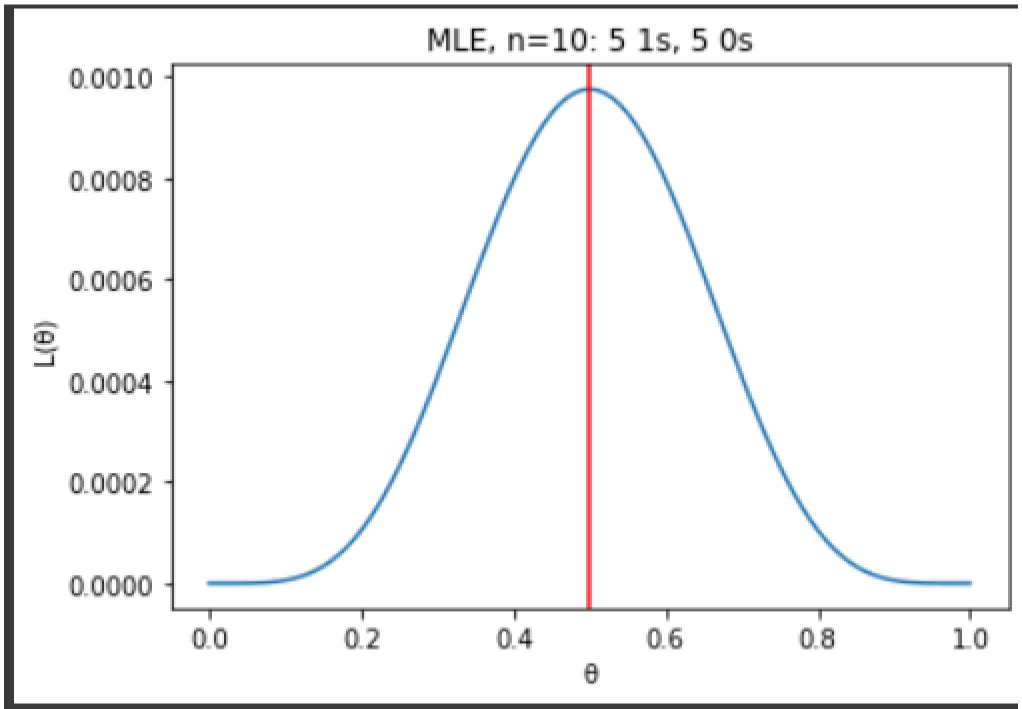
(d) Create three more likelihood plots: one where $n = 5$ and the data set contains three 1s and two 0s; one where $n = 100$ and the data set contains sixty 1s and forty 0s; and one where $n = 10$ and there are five 1s and five 0s. Include these plots in your writeup, and describe how the likelihood functions and maximum likelihood estimates compare for the different data sets.



```
Max likelihood is 0.03456 at θ=0.6
```



```
Max likelihood is 5.908465121038621e-30 at θ=0.6
```



```
Max likelihood is 0.0009765625 at θ=0.5
```

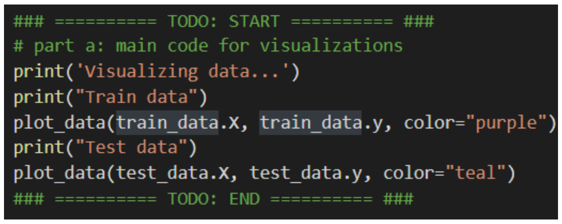
The MLE is equal to the proportion of $X_i = 1$ in the data, so for the 2 plots where the proportion of 1s is 60%, the MLE is 0.6, and for the last plot where the proportion of 1s is 50%, the MLE is 0.5.. As the amount of samples increases (n increases), L(θ) decreases, and the peak gets more narrow (the MLE becomes more clear).

4 Implementation: Polynomial Regression [40 pts]

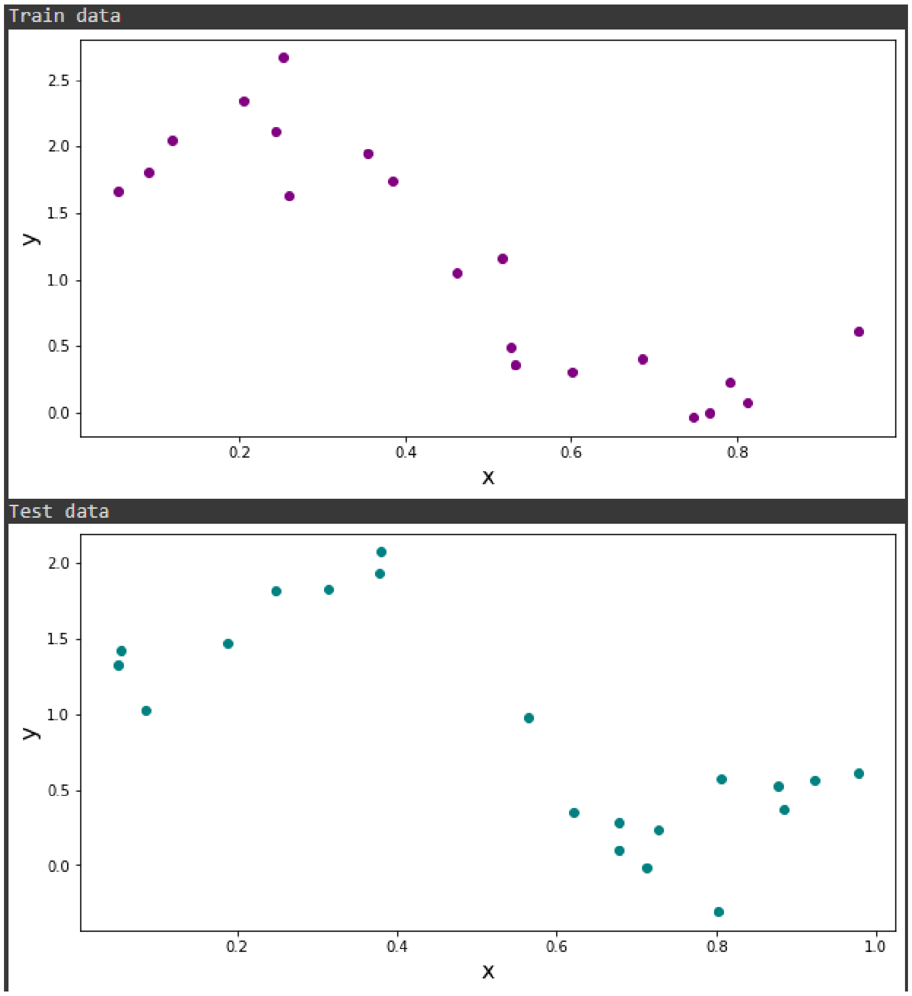
(a) Visualize the training and test data using the `plot_data(...)` function. What do you observe? For example, can you make an educated guess on the effectiveness of linear regression in predicting the data? [2 pts]

For the training data, linear regression may be somewhat effective at predicting the data because while the relationship is not completely linear, there is still a general trend. However, polynomial regression would be better.

For the test data, linear regression will be less effective because the data are even less linearly related. Overall, a polynomial fit will be better to predict the data.



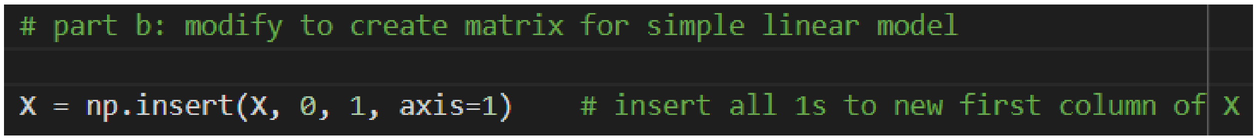
```
### ========== TODO: START ========== ###
# part a: main code for visualizations
print('Visualizing data...')
print("Train data")
plot_data(train_data.X, train_data.y, color="purple")
print("Test data")
plot_data(test_data.X, test_data.y, color="teal")
### ========== TODO: END ========== ###
```



```
Train data
```
```
Test data
```

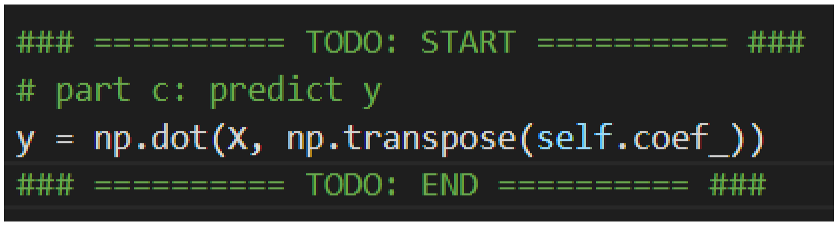
(b) Note that to take into account the intercept term (w_0), we can add an additional "feature" to each instance and set it to one, e.g. $x_{i,0} = 1$. This is equivalent to adding an additional first column to $\boldsymbol{X}$ and setting it to all ones. [2 pts].

Modify `PolynomialRegression.generate_polynomial_features(...)` to create the matrix $\boldsymbol{X}$ for a simple linear model. You do not need to turn in anything for this part.



```
# part b: modify to create matrix for simple linear model

X = np.insert(X, 0, 1, axis=1)    # insert all 1s to new first column of X
```

(c) Before tackling the harder problem of training the regression model, complete `PolynomialRegression.predict(...)` to predict $\boldsymbol{y}$ from $\boldsymbol{X}$ and $\boldsymbol{\theta}$. You do not need to turn in anything for this part. [3 pts]



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

(d) One way to solve linear regression is through gradient descent (GD).

Recall that the parameters of our model are the θ_j values. These are the values we will adjust to minimize $J(\boldsymbol{\theta})$.

$$J(\boldsymbol{\theta}) = \sum_{n=1}^{N} (h_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - y_n)^2$$

In gradient descent, each iteration performs the update

$$\theta_j \leftarrow \theta_j - 2\eta \sum_{n=1}^{N} (h_{\boldsymbol{\theta}}(\boldsymbol{x}_n) - y_n)\, x_{n,j} \quad \text{(simultaneously update } \theta_j \text{ for all } j\text{)}.$$

With each step of gradient descent, we expect our updated parameters θ_j to come closer to the parameters that will achieve the lowest value of $J(\boldsymbol{\theta})$. [10 pts]

- As we perform gradient descent, it is helpful to monitor the convergence by computing the cost, *i.e.*, the value of the objective function J. Complete `PolynomialRegression.cost(...)` to calculate $J(\boldsymbol{\theta})$.

  If you have implemented everything correctly, then the following code snippet should return 40.234 (you can print out the value of `model.cost` to confirm this).

  ```
  train_data = load_data(train_path)
  model = PolynomialRegression()
  model.coef_ = np.zeros(2)
  model.cost(train_data.X, train_data.y)
  ```

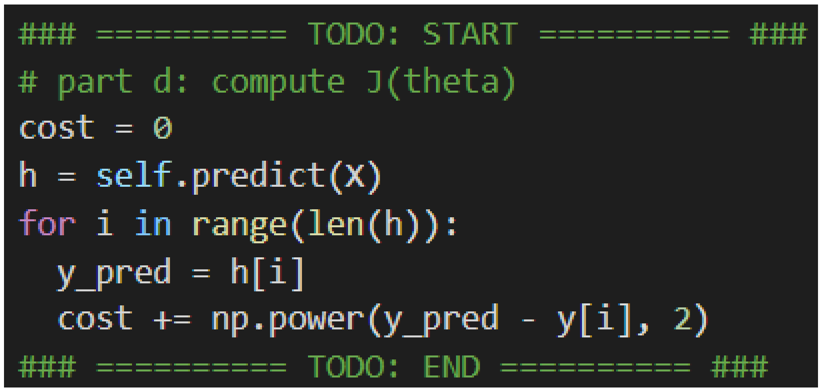
- Next, implement the gradient descent step in `PolynomialRegression.fit_GD(...)`. The loop structure has been written for you, and you only need to supply the updates to $\boldsymbol{\theta}$ and the new predictions $\hat{y} = h_{\boldsymbol{\theta}}(\boldsymbol{x})$ within each iteration.

  We will use the following specifications for the gradient descent algorithm:

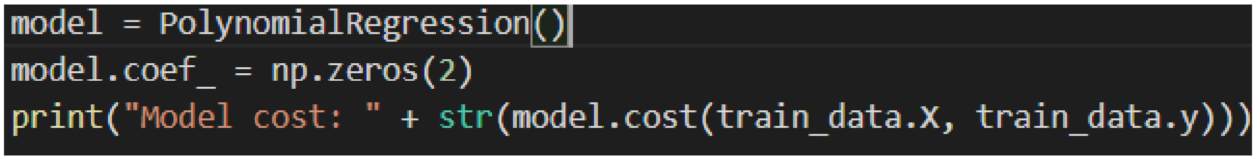
  - We run the algorithm for 10,000 iterations.
  - We terminate the algorithm earlier if the value of the objective function is unchanged across consecutive iterations.
  - We will use a fixed learning rate.

- Experiment with different values of learning rate $\eta = 10^{-6}, 10^{-5}, 10^{-3}, 0.05$ and make a table of the coefficients, number of iterations until convergence (this number will be 10,000 if the algorithm did not converge in a smaller number of iterations) and the final value of the objective function. How do the coefficients compare? How quickly does each algorithm converge? Do you observe something strange when you run with $\eta = 0.05$? Explain the observation and causes.

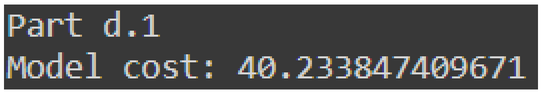
Part 1:



```
### ========== TODO: START ========== ###
# part d: compute J(theta)
cost = 0
h = self.predict(X)
for i in range(len(h)):
  y_pred = h[i]
  cost += np.power(y_pred - y[i], 2)
### ========== TODO: END ========== ###
```

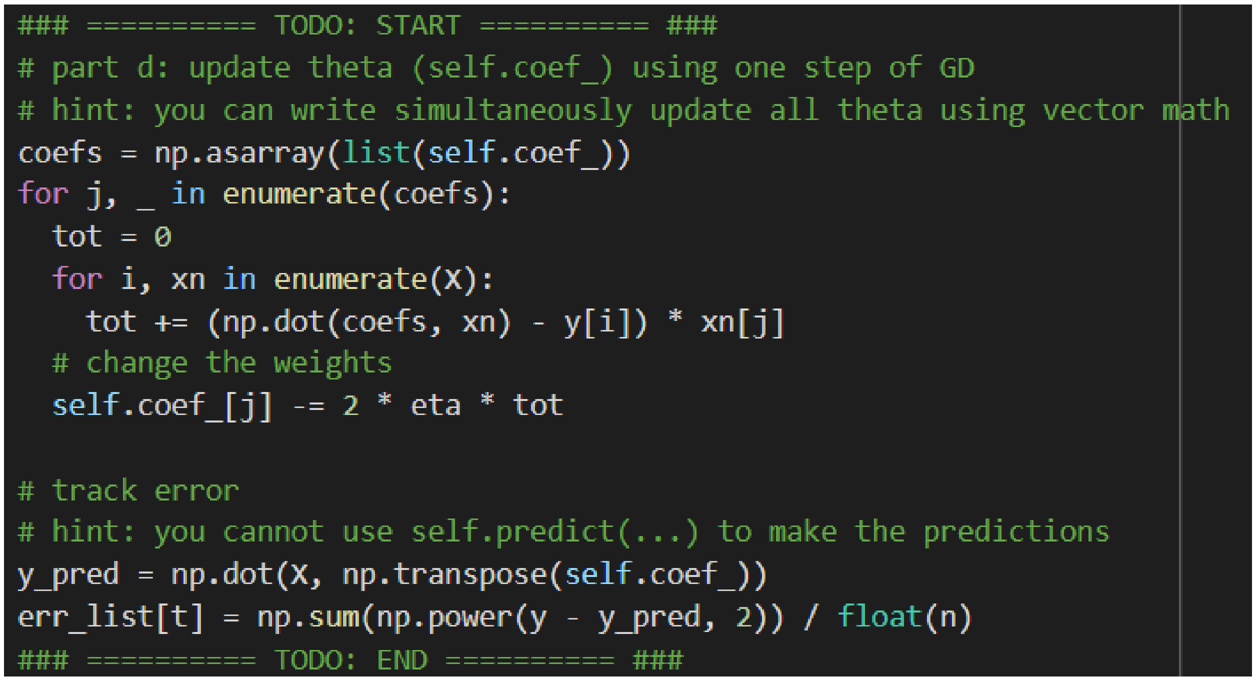


```
model = PolynomialRegression()
model.coef_ = np.zeros(2)
print("Model cost: " + str(model.cost(train_data.X, train_data.y)))
```



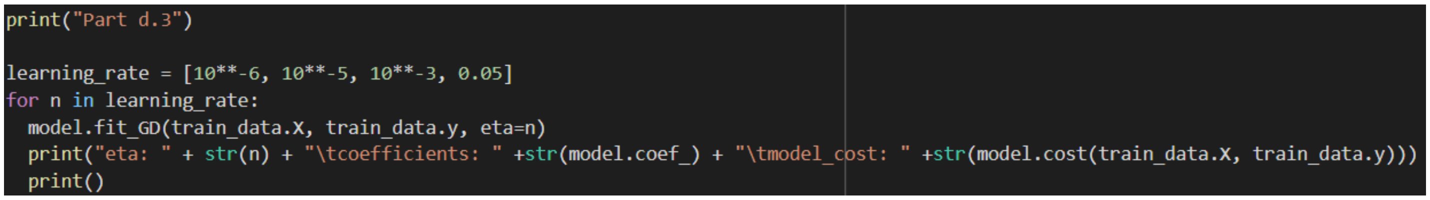
```
Part d.1
Model cost: 40.233847409671
```

Part 2:

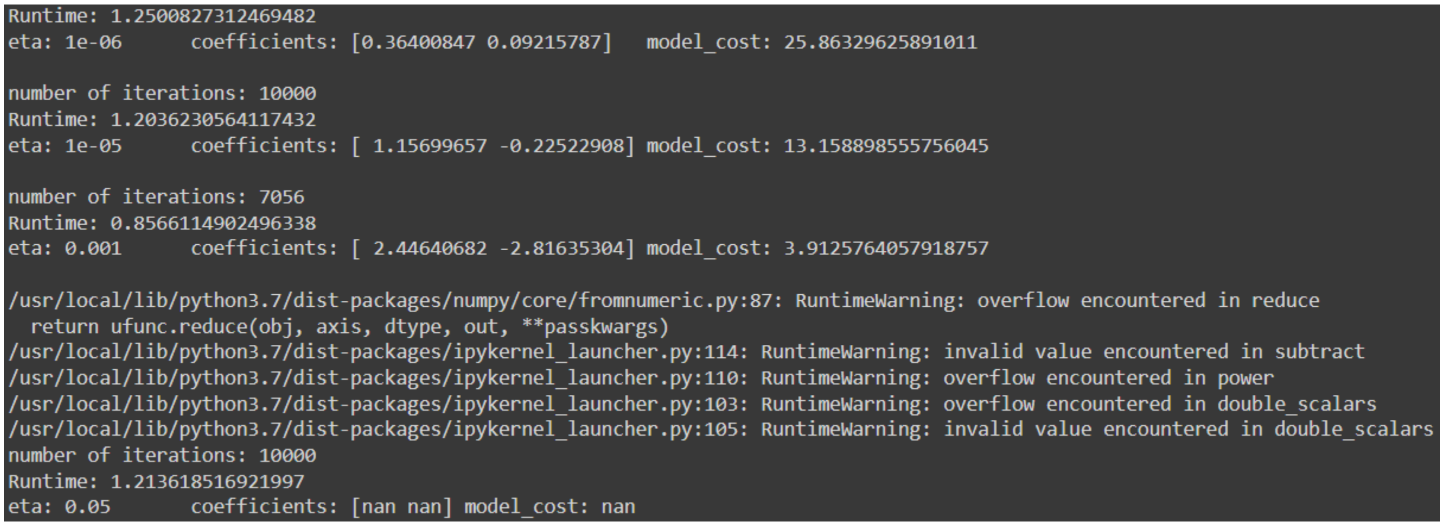


```python
### ========== TODO: START ========== ###
# part d: update theta (self.coef_) using one step of GD
# hint: you can write simultaneously update all theta using vector math
coefs = np.asarray(list(self.coef_))
for j, _ in enumerate(coefs):
  tot = 0
  for i, xn in enumerate(X):
    tot += (np.dot(coefs, xn) - y[i]) * xn[j]
  # change the weights
  self.coef_[j] -= 2 * eta * tot

# track error
# hint: you cannot use self.predict(...) to make the predictions
y_pred = np.dot(X, np.transpose(self.coef_))
err_list[t] = np.sum(np.power(y - y_pred, 2)) / float(n)
### ========== TODO: END ========== ###
```

Part 3:



```python
print("Part d.3")

learning_rate = [10**-6, 10**-5, 10**-3, 0.05]
for n in learning_rate:
  model.fit_GD(train_data.X, train_data.y, eta=n)
  print("eta: " + str(n) + "\tcoefficients: " +str(model.coef_) + "\tmodel_cost: " +str(model.cost(train_data.X, train_data.y)))
  print()
```



```
Runtime: 1.2500827312469482
eta: 1e-06      coefficients: [0.36400847 0.09215787]   model_cost: 25.86329625891011

number of iterations: 10000
Runtime: 1.2036230564117432
eta: 1e-05      coefficients: [ 1.15699657 -0.22522908] model_cost: 13.158898555756045

number of iterations: 7056
Runtime: 0.8566114902496338
eta: 0.001      coefficients: [ 2.44640682 -2.81635304] model_cost: 3.9125764057918757

/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:114: RuntimeWarning: invalid value encountered in subtract
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:110: RuntimeWarning: overflow encountered in power
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:103: RuntimeWarning: overflow encountered in double_scalars
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:105: RuntimeWarning: invalid value encountered in double_scalars
number of iterations: 10000
Runtime: 1.213618516921997
eta: 0.05       coefficients: [nan nan] model_cost: nan
```

Table of learning rates, coefficients, number of iterations until convergence, and cost function

Learning rate	θ_0	θ_1	Number of iterations	$J(\theta)$	Runtime (s)
10^{-6}	0.364	0.092	10000	25.863	1.25

10^{-5}	1.157	-0.225	10000	13.129	1.20
10^{-3}	2.446	-2.816	7056	3.9126	0.86
0.05	nan	nan	10000	nan	1.21

Very small learning rates lead to the algorithm not converging since the step sizes are too small to reach the local minimum in less than the max number of iterations, and so the cost function (error) is high and the coefficients are not close to what they converge to for more appropriate step sizes (θ_0 = 2.446, θ_0 = -2.816). As the learning rate increases, the cost decreases and the coefficients get closer to what they should be, and the algorithm actually converges for step size 10^{-3}. This is because the step size is not small enough to where the algorithm will not converge in less than 10000 iterations, and not too large to where at each step, the algorithm overshoots the local minimum, never reaching it.

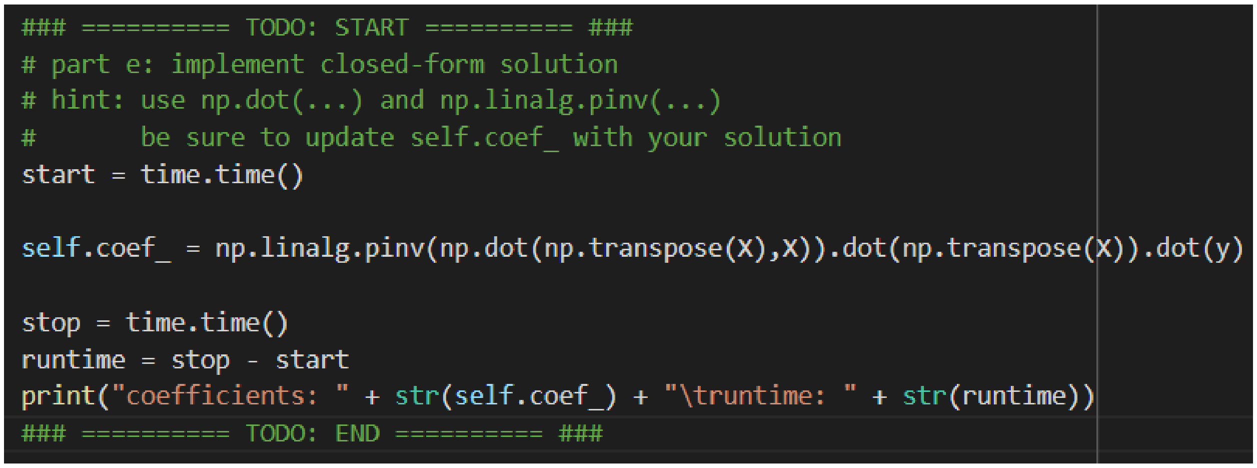
However, if the learning rate increases too much, as we see with 0.05, the algorithm does not converge. This is because at each step of gradient descent, due to the step size being too large, we overshoot the local minimum each time by jumping over it instead of converging to it.

(e) In class, we learned that the closed-form solution to linear regression is

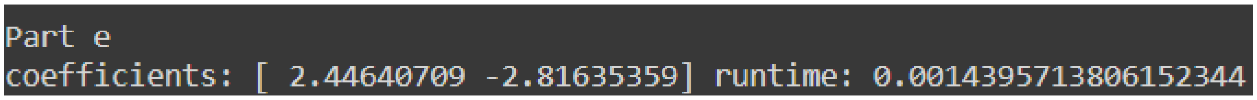
$$\boldsymbol{\theta} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}.$$

Using this formula, you will get an exact solution in one calculation: there is no "loop until convergence" like in gradient descent. [4 pts]

- Implement the closed-form solution `PolynomialRegression.fit(...)`.
- What is the closed-form solution coefficients? How do the coefficients and the cost compare to those obtained by GD? How quickly does the algorithm run compared to GD?



```
### ========== 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
start = time.time()

self.coef_ = np.linalg.pinv(np.dot(np.transpose(X),X)).dot(np.transpose(X)).dot(y)

stop = time.time()
runtime = stop - start
print("coefficients: " + str(self.coef_) + "\truntime: " + str(runtime))
### ========== TODO: END ========== ###
```



```
Part e
coefficients: [ 2.44640709 -2.81635359] runtime: 0.0014395713806152344
```

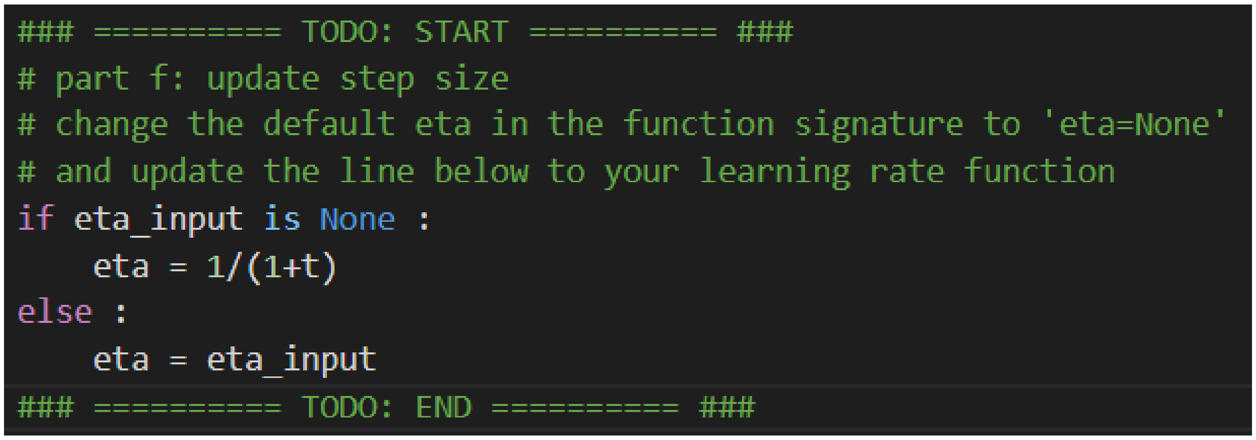
Closed form coefficients
θ_0 = 2.44640709, θ_1 = -2.81635359, cost function J: 3.9126
From part d.3
Gradient descent, step size 10^{-3}

Step size	θ_0	θ_1	Cost function	Runtime (s)
10^{-3}	2.44640682	-2.81635304	3.9126	0.86

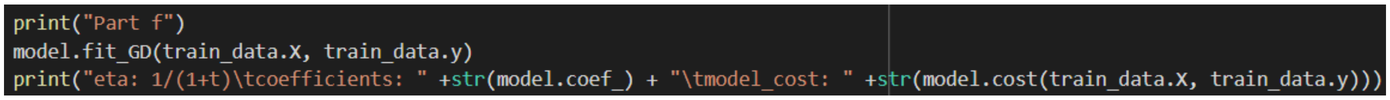
The closed form coefficients are nearly the same coefficients as the gradient descent coefficients
As for the runtime, the runtime for the closed-form solution is 0.0014 s, while the runtime for the gradient descent algorithm with step size 10^{-3} (which actually converges) has runtime 0.86 s. The closed form solution is over 600 times faster (much faster) than the gradient descent algorithm.

(f) Finally, set a learning rate η for GD that is a function of k (the number of iterations) (use $\eta_k = \frac{1}{1+k}$) and converges to the same solution yielded by the closed-form optimization (minus possible rounding errors). Update `PolynomialRegression.fit_GD(...)` with your proposed learning rate. How many iterations does it take the algorithm to converge with your proposed learning rate? [4 pts]

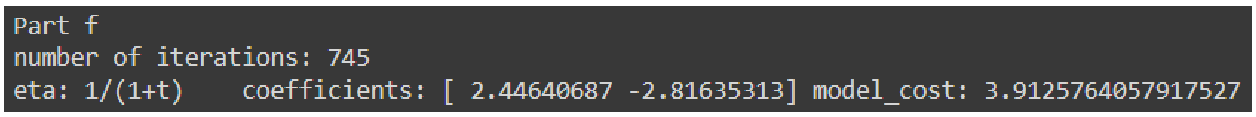
745 iterations



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



```
print("Part f")
model.fit_GD(train_data.X, train_data.y)
print("eta: 1/(1+t)\tcoefficients: " +str(model.coef_) + "\tmodel_cost: " +str(model.cost(train_data.X, train_data.y)))
```



```
Part f
number of iterations: 745
eta: 1/(1+t)    coefficients: [ 2.44640687 -2.81635313] model_cost: 3.9125764057917527
```

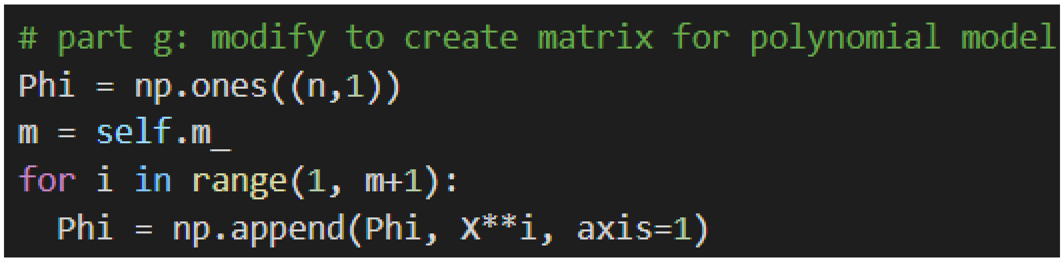
This is the same solution as the closed-form optimization solution, since we decrease the step size for each iteration so we step large at the beginning to converge under the max number of iterations and make small steps to converge at the closed form solution.

(g) Recall that polynomial regression can be considered as an extension of linear regression in which we replace our input matrix $\boldsymbol{X}$ with

$$\boldsymbol{\Phi} = \begin{pmatrix} \phi(x_1)^T \\ \phi(x_2)^T \\ \vdots \\ \phi(x_N)^T \end{pmatrix},$$

where $\phi(x)$ is a function such that $\phi_j(x) = x^j$ for $j = 0, \ldots, m$.

Update `PolynomialRegression.generate_polynomial_features(...)` to create an $m+1$ dimensional feature vector for each instance. You do not need to turn in anything for this part. [4 pts]



```
# part g: modify to create matrix for polynomial model
Phi = np.ones((n,1))
m = self.m_
for i in range(1, m+1):
  Phi = np.append(Phi, X**i, axis=1)
```

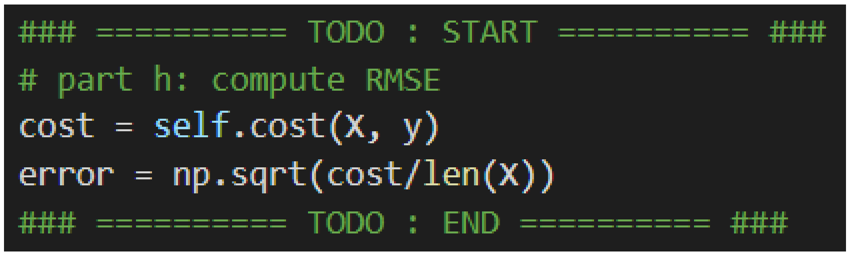
(h) Given N training instances, it is always possible to obtain a "perfect fit" (a fit in which all the data points are exactly predicted) by setting the degree of the regression to $N-1$. Of course, we would expect such a fit to generalize poorly. In the remainder of this problem, you will investigate the problem of overfitting as a function of the degree of the polynomial, m. To measure overfitting, we will use the Root-Mean-Square (RMS) error, defined as

$$E_{RMS} = \sqrt{J(\boldsymbol{\theta})/N},$$

where N is the number of instances.[4]

Why do you think we might prefer RMSE as a metric over $J(\boldsymbol{\theta})$?

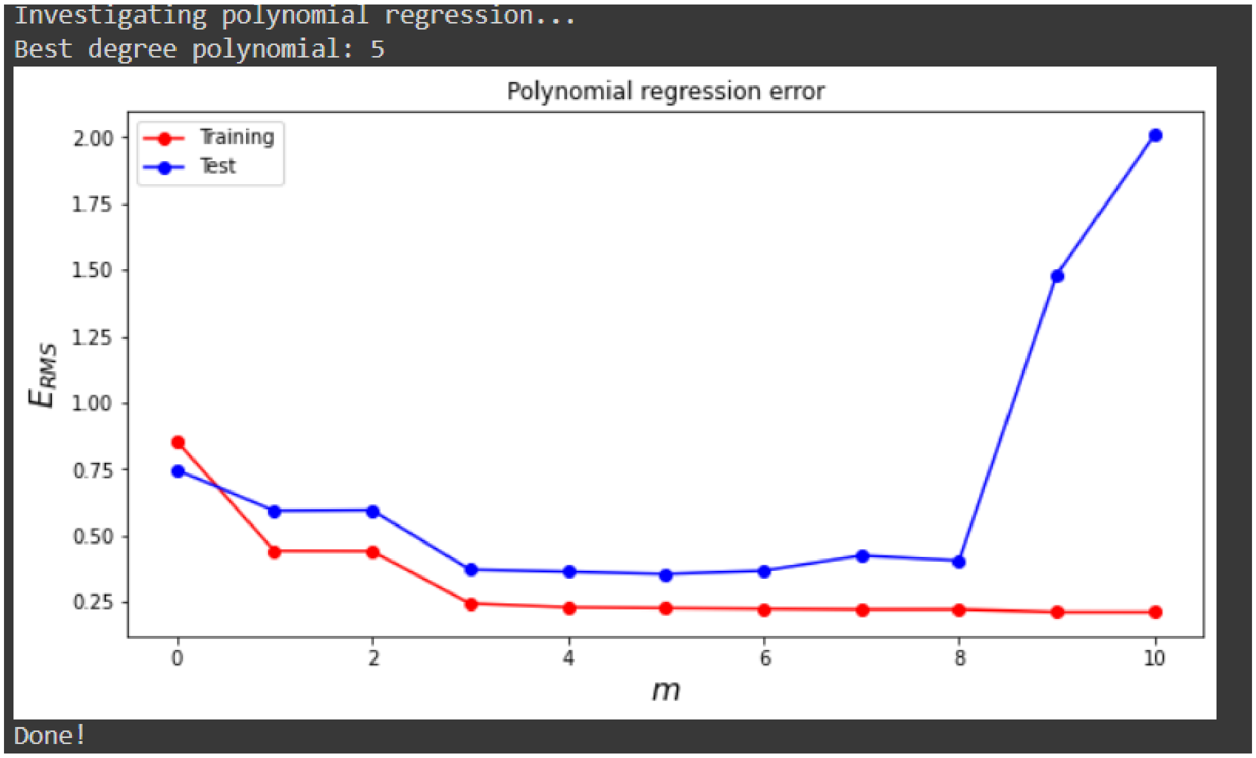
Implement `PolynomialRegression.rms_error(...)`. [4 pts]



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

RMSE is preferable to J(θ) because it is normalized by the number of training examples N. Since we use a normalized value RMSE, any error bias due to the size of the dataset is eliminated.

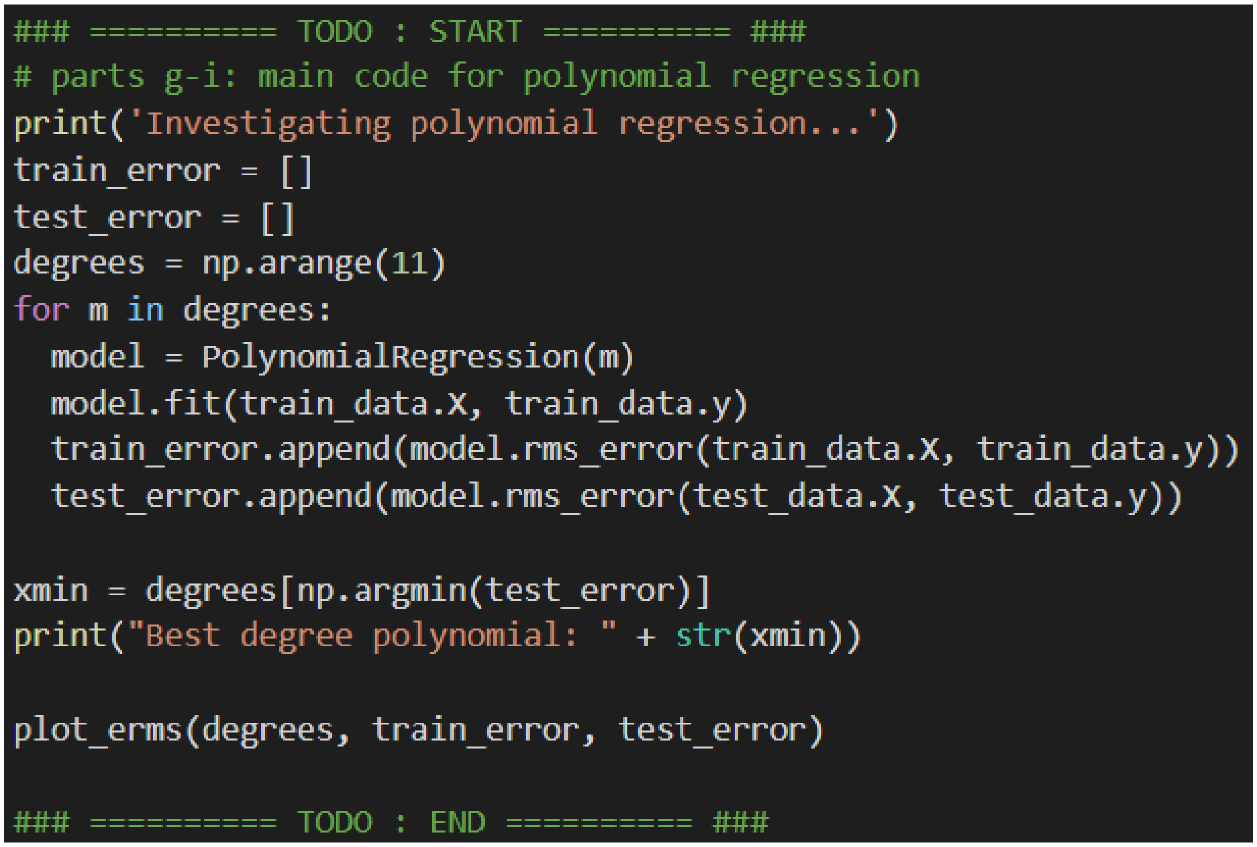
(i) For $m = 0, \ldots, 10$ (where m is the order of the polynomial transformation applied to features), use the closed-form solver to determine the best-fit polynomial regression model on the training data, and with this model, calculate the RMSE on both the training data and the test data. Generate a plot depicting how RMSE varies with model complexity (polynomial degree) – you should generate a single plot using `plot_erms(...)` function , and include this plot in your writeup. Which degree polynomial would you say best fits the data? Was there evidence of under/overfitting the data? Use your plot to justify your answer. [7 pts]



```
Investigating polynomial regression...
Best degree polynomial: 5
```

```
Done!
```

Polynomial degree m=3-6, with m=5 having the lowest test error, fits the data the best. There is clear evidence of overfitting, as after m=6, and most clearly seen at m=8, the test error gets worse and increases a lot, while the training error continues to get minimized. We also see evidence of underfitting, as at m=0, the training error is worse than the test error, meaning that the model is too simple, and at m < 3, the training error is not minimized.



```
### ========== TODO : START ========== ###
# parts g-i: main code for polynomial regression
print('Investigating polynomial regression...')
train_error = []
test_error = []
degrees = np.arange(11)
for m in degrees:
  model = PolynomialRegression(m)
  model.fit(train_data.X, train_data.y)
  train_error.append(model.rms_error(train_data.X, train_data.y))
  test_error.append(model.rms_error(test_data.X, test_data.y))

xmin = degrees[np.argmin(test_error)]
print("Best degree polynomial: " + str(xmin))

plot_erms(degrees, train_error, test_error)

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