Problem Set 2

Graded

Student

BRANDON LO

Total Points

42.9 / 47 pts

Question 1

Perceptron **1.9** / 2 pts

1.1 part (a) OR **0.9** / 1 pt

✔ **– 0.1 pts** **Incorrect**: Assumed/accidentally demonstrated that False is 0 and not -1; however, this was such a common mistake that most credit is given

1.2 part (b) XOR **1** / 1 pt

✔ **– 0 pts** **Correct**: Not linearly separable; no perceptron exists

Question 2

Logistic Regression **7** / 10 pts

2.1 part (a) Partial Derivatives **2** / 2 pts

✔ **– 0 pts** **Correct**: $\frac{\partial J}{\partial \theta_j} = \sum_{n=1}^{N} (h_\theta(\mathbf{x}^{(n)}) - y_n) x_j^{(n)}$

2.2 part (b) Hessian **3** / 3 pts

✔ **– 0 pts** **Correct**: $\mathbf{H} = \sum_{n=1}^{N} h_\theta(\mathbf{x}^{(n)})(1 - h_\theta(\mathbf{x}^{(n)}))\mathbf{x}^{(n)}\mathbf{x}^{(n)^T}$

2.3 part (c) Convex Function **2** / 5 pts

✔ **– 1 pt** **Partial**: Doesn't show ≥ 0 for all $\mathbf{z} \in \mathbb{R}^d$

✔ **– 2 pts** **Partial**: Attempted to simplify/expand $\mathbf{z}^T \mathbf{H} \mathbf{z}$; final representation is wrong

Question 3

Maximum Likelihood Estimation **14** / 15 pts

3.1 **part (a) Likelihood Function** **3** / 3 pts

✔ **– 0 pts** Correct

3.2 **part (b) MLE** **5** / 6 pts

✔ **– 1 pt** Does not show the function is convex/concave

3.3 **part (c) Likelihood plot Theta** **3** / 3 pts

✔ **– 0 pts** Correct

3.4 **part (d) Likelihood plot Data** **3** / 3 pts

✔ **– 0 pts** Correct

Question 4

Linear and Polynomial Regression **20** / 20 pts

4.1 **part (a) Visualization** **1** / 1 pt

✔ **– 0 pts** Correct

4.2 **part (b) Constructing X (code snippets)** **1** / 1 pt

✔ **– 0 pts** correct if part(d) is correct

4.3 **part (c) Prediction (code snippets)** **1** / 1 pt

✔ **– 0 pts** correct if part(d) is correct

4.4 **part (d) Gradient Descent** **5** / 5 pts

✔ **– 0 pts** Correct

4.5 **part (e) Close Form** **4** / 4 pts

✔ **– 0 pts** Correct

4.6 **part (f) Auto Learning Rate** **1** / 1 pt

✔ **– 0 pts** Correct

4.7 **part (g) Constructing Polynomial X (code snippets)** **1** / 1 pt

✔ **– 0 pts** Correct

4.8 **part (h) RMSE (code snippets)** **2** / 2 pts

✔ **– 0 pts** Correct

4.9 **part (i) Model Complexity** **4** / 4 pts

✔ **– 0 pts** Correct

Questions assigned to the following page: 1.2 and 1.1

1 Perceptron [2 pts]

Design (specify $\boldsymbol{\theta}$ 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) **(1 pts)** OR

(b) **(1 pts)** XOR

For inputs X_1 and X_2, output Y

a. OR:

X_1	X_2	Y
0	0	0
1	0	1
0	1	1
1	1	1

Two input perceptron:

$X_1 + X_2 + b$, where $b = -\frac{1}{2}$

b. XOR:

X_1	X_2	Y
0	0	0
1	0	1
0	1	1
0	0	0

There is no two input perceptron that can compute XOR because the data is not linearly separable.

Questions assigned to the following page: 2.2 and 2.1

2 Logistic Regression [10 pts]

Consider the objective function that we minimize in logistic regression:

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

where $h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \sigma(\boldsymbol{\theta}^T \boldsymbol{x}) = \frac{1}{1+e^{-\boldsymbol{\theta}^T \boldsymbol{x}}}$.

(a) **(2 pts)** Find the partial derivatives $\frac{\partial J}{\partial \theta_j}$.

(b) **(3 pts)** 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 $\boldsymbol{H} = \sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x}_n)(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x}_n))\boldsymbol{x}_n \boldsymbol{x}_n^T$.

(c) **(5 pts)** 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 $\boldsymbol{z}$.

a. Using the chain rule: $\frac{d}{d\theta_j} h_\theta(x) = h_\theta(x)(1-h_\theta(x))\, x_j$

We can then say: $\frac{dJ}{d\theta_j} = -\sum_{n=1}^{N} \left[y_n \frac{1}{h_\theta(x_n)} - (1-y_n) \frac{1}{1-h_\theta(x_n)} \right] \frac{d}{d\theta_j} h_\theta(x_n)$

$$\frac{dJ}{d\theta_j} = -\sum_{n=1}^{N} \left[y_n \frac{1}{h_\theta(x_n)} - (1-y_n) \frac{1}{1-h_\theta(x_n)} \right] h_\theta(x)(1-h_\theta(x))\, x_{n,j}$$

Distribute: $\frac{dJ}{d\theta_j} = -\sum_{n=1}^{N} \left[y_n (1-h_\theta(x_n)) - (1-y_n) h_\theta(x_n) \right] x_{n,j}$

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

b. We differentiate our answer from part a to find $\frac{d^2 J}{d\theta_j d\theta_k}$

$$\frac{d^2 J}{d\theta_j d\theta_k} = \frac{d}{d\theta_k} \left(\frac{dJ}{d\theta_j} \right)$$

$$\frac{d^2 J}{d\theta_j d\theta_k} = \frac{d}{d\theta_k} \left(\sum_{n=1}^{N} [h_\theta(x_n) - y_n]\, x_{n,j} \right)$$

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

Therefore, Hessian matrix H with elements H_{jk} is given by:

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

Question assigned to the following page: 2.3

C. A function is convex if its Hessian is positive semi-definite, which means this condition holds:

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

This means that each element H_{jk} is non-negative. From our answer from part b, we know each element H_{jk} is non-negative because $h_\theta(x_n)(1-h_\theta(x_n))$ is a product of probabilities. Therefore $\sum_{j,k} z_j z_k H_{jk}$ is non negative meaning H is positive semi-definite and convex.

Question assigned to the following page: 3.1

3 Maximum Likelihood Estimation [15 pts]

Suppose we observe the values of n independent random variables $X_1, \ldots, X_n$ 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 $X_1, \ldots, X_n$ 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) **(3 pts)** Write a formula for the likelihood function, $L(\theta) = P(X_1, \ldots, X_n; \theta)$. Your function should depend on the random variables $X_1, \ldots, X_n$ and the hypothetical parameter θ. Does the likelihood function depend on the order in which the random variables are observed ?

(b) **(6 pts)** 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. For this part, assume we use the natural logarithm (i.e. logarithm base e).

(c) **(3 pts)** 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 θ 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 θ that maximizes the likelihood. Does the answer agree with the closed form answer ?

(d) **(3 pts)** 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.

a. For Bernoulli distribution, assuming independent and identically distributed data, the likelihood function is a product of probabilities.

$$L(\theta) = \prod_{n=1}^{N} P(x_n; \theta)$$

To be more specific, we can write this as the product of each individual outcome.

$$L(\theta) = \prod_{n=1}^{N} \theta^{x_n} (1-\theta)^{1-x_n}$$

We assume random variables are independent and identically distributed, so the order in which they are observed does not matter.

Questions assigned to the following page: 3.2 and 3.3

b. To find log-likelihood, we take the log of our product of probabilities from part a.

$$l(\theta) = \log\left(\prod_{n=1}^{N} P(x_n;\theta)\right) = \sum_{n=1}^{N} \log\left(P(x_n;\theta)\right) = \sum_{n=1}^{N} \left[x_n \log\theta + (1-x_n)\log(1-\theta)\right]$$

To maximize this function, we find the derivative and set it equal to 0.

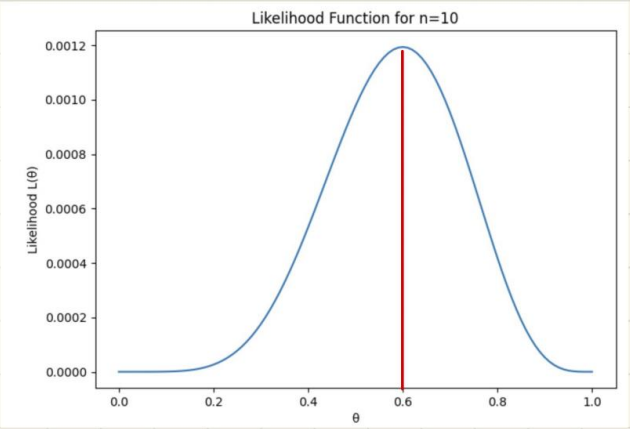
$$\frac{dl}{d\theta} = \sum_{n=1}^{N} \left[\frac{x_n}{\theta} - \frac{1-x_n}{1-\theta}\right] = 0$$

$$\theta = \frac{\Sigma x_n}{N} = \bar{X}$$

To verify concavity, we can find the second derivative:

$$\frac{d^2l}{d\theta^2} = \sum_{n=1}^{N} \left[-\frac{x_n}{\theta^2} - \frac{1-x_n}{(1-\theta)^2}\right]$$

c.

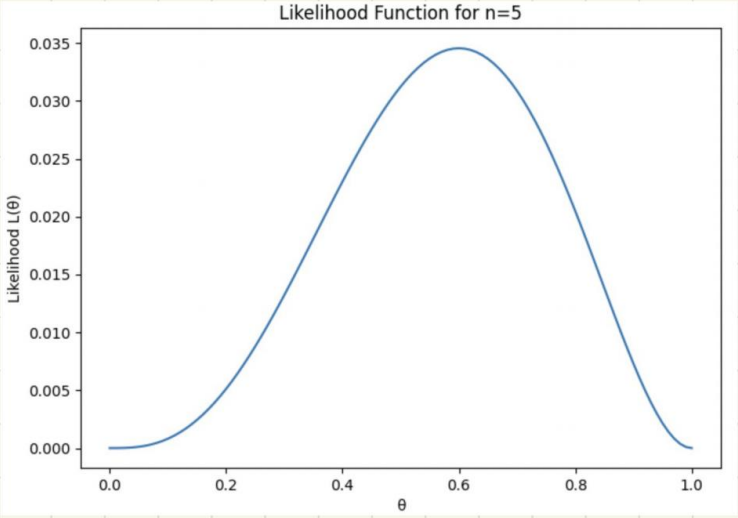


Six 1s Four 0s

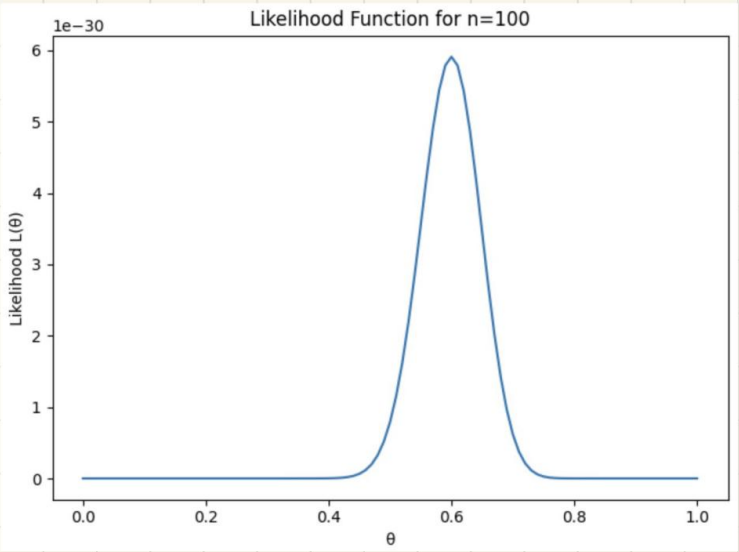
The value of θ which maximizes the likelihood is 0.6. This is consistent with the solution from part b where we set the first derivative of log-likelihood equal to zero.

Question assigned to the following page: 3.4

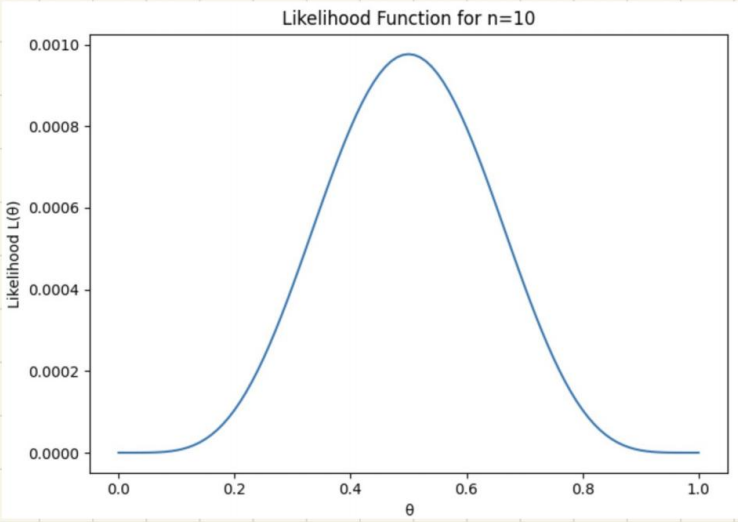
d.



Three 1s and two 0s:
Graph centered at 0.6 due to ratio of successes to failures, the standard deviation is large indicating a wider range of likelihood estimates.



Sixty 1s and forty 0s:
Graph is still centered at 0.6 as the ratio remains the same. The standard deviation is smaller (graph is narrower) due to larger sample size.



Five 1s and five 0s:
Graph is centered at 0.5 due to equal number of 1s and 0s. The standard deviation is similar to the other n=10 graph but slightly wider.

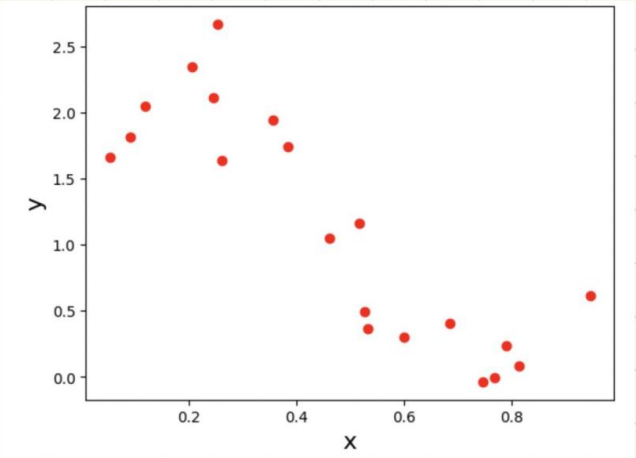
Question assigned to the following page: 4.1

Visualization [1 pts]

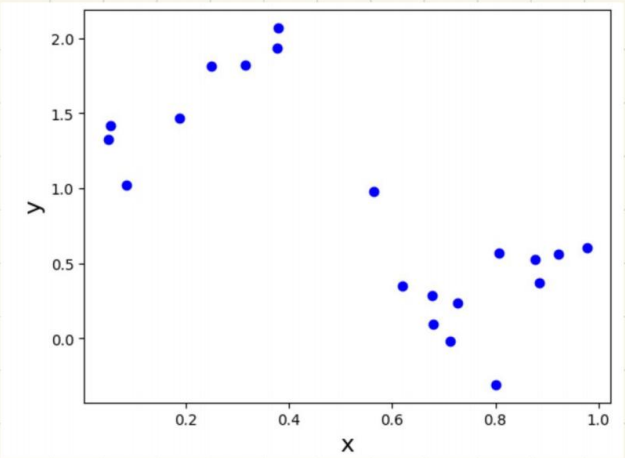
It is often useful to understand the data through visualizations. For this data set, you can use a scatter plot to visualize the data since it has only two properties to plot (x and y).

(a) **(1 pts)** 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?

Training Data:



Test Data:



Both datasets exhibit significant amount of noise. However, the test data exhibits more. The training data shows negative correlation between X and Y. These characteristics suggest polynomial regression may be a better fit for the test data than linear regression.

Questions assigned to the following page: 4.2 and 4.3

Linear Regression [12 pts]

Recall that linear regression attempts to minimize the objective function

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

In this problem, we will use the matrix-vector form where

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \qquad \boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_N^T \end{pmatrix}, \qquad \boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_D \end{pmatrix}$$

and each instance $\boldsymbol{x}_n = \left(1, x_{n,1}, \ldots, x_{n,D}\right)^T$.

In this instance, the number of input features $D = 1$.

Rather than working with this fully generalized, multivariate case, let us start by considering a simple linear regression model:

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta}^T \boldsymbol{x} = \theta_0 + \theta_1 x_1$$

The Colab notebook contains the skeleton code for the class `PolynomialRegression`. Objects of this class can be instantiated as `model = PolynomialRegression (m)` where m is the degree of the polynomial feature vector where the feature vector for instance n is $\left(1, x_{n,1}, x_{n,1}^2, \ldots, x_{n,1}^m\right)^T$. Setting $m = 1$ instantiates an object where the feature vector for instance n is $\left(1, x_{n,1}\right)^T$.

(b) **(1 pts)** Note that to take into account the intercept term (θ_0), we can add an additional "feature" to each instance and set it to one, e.g. $x_{n,0} = 1$. This is equivalent to adding an additional first column to $\boldsymbol{X}$ and setting it to all ones.

Modify `PolynomialRegression.generate_polynomial_features(...)` to create the matrix $\boldsymbol{X}$ for a simple linear model. Copy-paste/screenshot your code snippet.

```
# part b: modify to create matrix for simple linear model
X = np.append(np.ones([n,1]), X, 1)
```

(c) **(1 pts)** Before tackling the harder problem of training the regression model, complete `PolynomialRegression.predict(...)` to predict $\boldsymbol{y}$ from $\boldsymbol{X}$ and $\boldsymbol{\theta}$. Copy-paste/screenshot your code snippet.

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

Question assigned to the following page: 4.4

(d) **(5 pts)** 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 $J(\boldsymbol{\theta})$.

- 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.

```
train_data = load_data('regression_train.csv') # Use your own 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 $\eta = 10^{-4}, 10^{-3}, 10^{-2}, 0.1$, 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?

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

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

```
# part d: compute J(theta)
cost = 0

h = self.predict(X)
for ind, j in enumerate(h):
  cost += (h[ind] - y[ind]) ** 2
```

These are the results from running the code:

```
number of iterations: 10000
eta: 0.0001     model.coef_:[ 2.27044798 -2.46064834]   model_cost:4.086397036795765
number of iterations: 7020
eta: 0.001      model.coef_:[ 2.4464068 -2.816353 ]     model_cost:3.9125764057919463
number of iterations: 764
eta: 0.01       model.coef_:[ 2.44640703 -2.81635346]   model_cost:3.9125764057914862
number of iterations: 10000
eta: 0.1        model.coef_:[nan nan]   model_cost:nan
```

Question assigned to the following page: 4.4

The step size of 10^{-2} yielded the best outcome of efficiency and number of iterations.
The step size of 10^{-3} still converged before 10,000 iterations but had a slightly higher model cost.
The step size of 10^{-9} was too small and did not converge within 10,000 iterations.
The step size of 0.1 was too large and did not converge within 10,000 iterations. The coefficients are also numerically unstable which means the line is likely incorrect.

Question assigned to the following page: 4.5

(e) **(4 pts)** 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.

- 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? Use the 'time' module to compare its runtime to GD.

```
# part e: implement closed-form solution
# hint: use np.dot(...) and np.linalg.pinv(...)
#       be sure to update self.coef_ with your solution

print("part e: ")
start_time = time.time()
self.coef_ = (np.linalg.pinv(np.dot(np.transpose(X),X)).dot(np.transpose(X))).dot(y)
stop_time = time.time()
run_time = stop_time - start_time
print(f"coefficient:{self.coef_}\trun time:{run_time}")

return self
```

Results:

```
part e:
coefficient:[ 2.44640709 -2.81635359]   run time:0.0017116069793701172
```

The coefficients are the closed-form solution coefficients. The coefficients and cost of 3.912576 are more precise and more efficient than those obtained by gradient descent. The 0.00117 seconds is also much faster than the 0.12098 seconds of gradient descent.

Questions assigned to the following page: 4.6, 4.7, and 4.8

(f) **(1 pts)** 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?

```
# 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 = float(1)/(1+t)
 else :
    eta = eta_input
```

The algorithm converges at 1511 iterations with my proposed learning rate

Polynomial Regression [7 pts]

Now let us consider the more complicated case of polynomial regression, where our hypothesis is

$$h_{\boldsymbol{\theta}}(\boldsymbol{x}) = \boldsymbol{\theta}^T \phi(\boldsymbol{x}) = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_m x^m.$$

(g) **(1 pts)** 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. Copy-paste/screenshot your code snippet.

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

(h) **(2 pts)** 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]

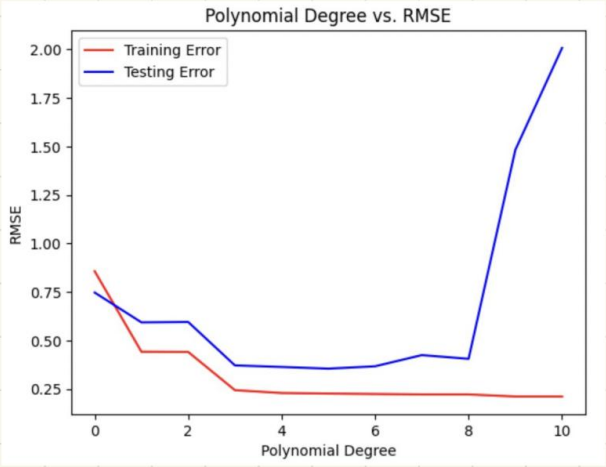
Why do you think we might prefer RMSE as a metric over $J(\boldsymbol{\theta})$?

Implement `PolynomialRegression.rms_error(...)`. Copy-paste/screenshot your code snippet.

```
# part h: compute RMSE
error = 0
error = np.sqrt(self.cost(X, y)/X.shape[0])
```

Question assigned to the following page: 4.9

(i) **(4 pts)** For $m = 0, \ldots, 10$ (where m is the degree of the polynomial), 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 with both training and test error, 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.



When $m<3$ the model is underfitting due to relatively large training and test error, most prevalant where $m<1$ and training error is above test error.
Will overfit when $m>8$ as shown by large testing error and low training error.
The best fit for the data is where both errors are lowest, which looks to be where $m=5$ (or at least $4<m<6$).