Problem Set 2: Perceptron and Regression

Bonnie Liu
UID: 005300989
Discussion 1A (Ulzee An)

Due Date: February 12, 2022

For this problem set, I collaborated with Henry Li, Hannah Zhong, David Xiong, Justin He, and Nicholas Dean.

1 Perceptron

1.1 OR

Assuming T = 1 and F = -1, I came up with the following perceptron for OR:

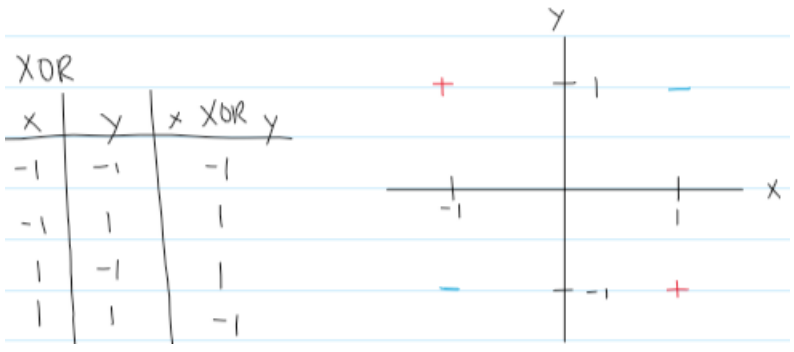
$$\boldsymbol{\theta} = \begin{pmatrix} 30 \\ 20 \\ 20 \end{pmatrix}$$

Another possible perceptron with a different hyperplane for OR is:

$$\boldsymbol{\theta} = \begin{pmatrix} 25 \\ 20 \\ 20 \end{pmatrix}$$

1.2 XOR

No perceptron exists for XOR because no line can classify this model, as seen below:



2 Logistic Regression

2.1

First, let's rewrite our objective function:

$$
\begin{aligned}
J(\boldsymbol{\theta}) &= -\sum_{n=1}^{N}[y_n log((1+e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}})^{-1}) + (1-y_n)log(1-\frac{1}{1+e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}})] \\
&= -\sum_{n=1}^{N}[-y_n log(1+e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}) + (1-y_n)log(\frac{e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}}{1+e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}})] \\
&= -\sum_{n=1}^{N}[-y_n log(1+e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}) + (1-y_n)log(\frac{e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}}{1+e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}})] \\
&= -\sum_{n=1}^{N}[-y_n log(1+e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}) + (1-y_n)log(e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}) - (1-y_n)log(1+e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}})] \\
&= -\sum_{n=1}^{N}[-\boldsymbol{\theta}^T\boldsymbol{x_n} + \boldsymbol{\theta}^T\boldsymbol{x_n}y_n - log(1+e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}] \\
&= -\sum_{n=1}^{N}[\boldsymbol{\theta}^T\boldsymbol{x_n}y_n - (log(e^{\boldsymbol{\theta}^T\boldsymbol{x_n}}) + log(1+e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}))] \\
&= -\sum_{n=1}^{N}[\boldsymbol{\theta}^T\boldsymbol{x_n}y_n - log(e^{\boldsymbol{\theta}^T\boldsymbol{x_n}}(1+e^{-\boldsymbol{\theta}^T\boldsymbol{x_n}}))] \\
&= -\sum_{n=1}^{N}[\boldsymbol{\theta}^T\boldsymbol{x_n}y_n - log(1+e^{\boldsymbol{\theta}^T\boldsymbol{x_n}})]
\end{aligned}
$$

Now let's take the derivative with respect to θ_j:

$$
\begin{aligned}
\frac{\partial J}{\partial \theta_j} &= -\sum_{n=1}^{N}[y_n x_{n,j} - \frac{x_{n,j}e^{\boldsymbol{\theta}^T\boldsymbol{x_n}}}{1+e^{\boldsymbol{\theta}^T\boldsymbol{x_n}}}] \\
&= -\sum_{n=1}^{N}(y_n - \frac{e^{\boldsymbol{\theta}^T\boldsymbol{x_n}}}{1+e^{\boldsymbol{\theta}^T\boldsymbol{x_n}}})x_{n,j} \\
&= -\sum_{n=1}^{N}(y_n - \frac{1}{1+e^{\boldsymbol{\theta}^T\boldsymbol{x_n}}})x_{n,j} \\
&= -\sum_{n=1}^{N}(y_n - h_{\boldsymbol{\theta}}(\boldsymbol{x_n}))x_{n,j} \\
&= \sum_{n=1}^{N}(h_{\boldsymbol{\theta}}(\boldsymbol{x_n}) - y_n)x_{n,j}
\end{aligned}
$$

2.2

Let's begin by finding the second partial derivatives. Recall the following from class: $\frac{d\sigma(a)}{da} = \sigma(a)(1-\sigma(a))$. Then

$$\begin{aligned}
\frac{\partial^2 J}{\partial\theta_j \partial\theta_k} &= \frac{\partial}{\partial\theta_k}[\sum_{n=1}^{N}(h_{\boldsymbol{\theta}}(\boldsymbol{x_n}) - y_n)x_{n,j}] \\
&= \sum_{n=1}^{N} x_{n,j}\frac{\partial}{\partial\theta_k}\sigma(\boldsymbol{\theta}^T\boldsymbol{x_n}) \\
&= \sum_{n=1}^{N} x_{n,j}\sigma(\boldsymbol{\theta}^T\boldsymbol{x_n})(1-\sigma(\boldsymbol{\theta}^T\boldsymbol{x_n}))x_{n,k} \\
&= \sum_{n=1}^{N} x_{n,j}x_{n,k}h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n}))
\end{aligned}$$

The Hessian can be written as $\boldsymbol{H} = \sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n}))\boldsymbol{x_n}\boldsymbol{x_n}^T$ because recall the definition of the Hessian matrix:

$$\begin{aligned}
\boldsymbol{H} &= \begin{pmatrix} \frac{\partial^2 J}{\partial\theta_1^2} & \cdots & \frac{\partial^2 J}{\partial\theta_1\partial\theta_k} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 J}{\partial\theta_k\partial\theta_1} & \cdots & \frac{\partial^2 J}{\partial\theta_k^2} \end{pmatrix} \\
&= \begin{pmatrix} \sum_{n=1}^{N} x_{n,1}^2 h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n})) & \cdots & \sum_{n=1}^{N} x_{n,1}x_{n,k}h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n})) \\ \vdots & \ddots & \vdots \\ \sum_{n=1}^{N} x_{n,k}x_{n,1}h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n})) & \cdots & \sum_{n=1}^{N} x_{n,k}^2 h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n})) \end{pmatrix}
\end{aligned}$$

Our goal is to show $\boldsymbol{H} = \sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n}))\boldsymbol{x_n}\boldsymbol{x_n}^T$, so let's expand the expression.

$$\begin{aligned}
\boldsymbol{H} &= \sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n}))\boldsymbol{x_n}\boldsymbol{x_n}^T \\
&= \sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n})) \begin{pmatrix} x_{n,1}^2 & \cdots & x_{n,1}x_{n,k} \\ \vdots & \ddots & \vdots \\ x_{n,k}x_{n,1} & \cdots & x_{n,k}^2 \end{pmatrix} \\
&= \begin{pmatrix} \sum_{n=1}^{N} x_{n,1}^2 h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n})) & \cdots & \sum_{n=1}^{N} x_{n,1}x_{n,k}h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n})) \\ \vdots & \ddots & \vdots \\ \sum_{n=1}^{N} x_{n,k}x_{n,1}h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n})) & \cdots & \sum_{n=1}^{N} x_{n,k}^2 h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1-h_{\boldsymbol{\theta}}(\boldsymbol{x_n})) \end{pmatrix}
\end{aligned}$$

We see we get the same matrix we obtain from using the definition of the Hessian matrix.

2.3

A function J is convex if its Hessian is positive semi-definite (PSD), written $\boldsymbol{H} \geq 0$. 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 $\boldsymbol{z}$. This means we need to show $\boldsymbol{z}^T \boldsymbol{H} \boldsymbol{z} \geq 0$ for any vector $\boldsymbol{z}$.

$$\begin{aligned}
\boldsymbol{z}^T \boldsymbol{H} \boldsymbol{z} &= \boldsymbol{z}^T (\sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x_n})) \boldsymbol{x_n} \boldsymbol{x_n}^T) \boldsymbol{z} \\
&= \sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x_n})) \boldsymbol{z}^T \boldsymbol{x_n} \boldsymbol{x_n}^T \boldsymbol{z} \\
&= \sum_{n=1}^{N} h_{\boldsymbol{\theta}}(\boldsymbol{x_n})(1 - h_{\boldsymbol{\theta}}(\boldsymbol{x_n}))(\boldsymbol{x_n}^T \boldsymbol{z})^T (\boldsymbol{x_n}^T \boldsymbol{z})
\end{aligned}$$

$(\boldsymbol{x_n}^T \boldsymbol{z})^T (\boldsymbol{x_n}^T \boldsymbol{z}) \geq 0$ because $i^2 \geq 0$ for all real i. Since $h_{\boldsymbol{\theta}}(\boldsymbol{x_n})$ is defined by the logistic function, $0 \leq h_{\boldsymbol{\theta}}(\boldsymbol{x_n}) \leq 1$. It follows that $0 \leq 1 - h_{\boldsymbol{\theta}}(\boldsymbol{x_n}) \leq 1$. Since $\boldsymbol{z}^T \boldsymbol{H} \boldsymbol{z}$ is the summation of the product of the aforementioned terms, $\boldsymbol{z}^T \boldsymbol{H} \boldsymbol{z} \geq 0$. Therefore, J is a convex function and thus has no local minima other than the global one.

3 Maximum Likelihood Estimation

3.1

$$\begin{aligned}
L(\theta) &= P(X_1, ..., X_n; \theta) \\
&= \prod_{i=1}^{n} [X_i \theta + (1 - X_i)(1 - \theta)] \\
&= \prod_{i=1}^{n} [\theta^{X_i} (1 - \theta)^{1 - X_i}]
\end{aligned}$$

No, the likelihood function does not depend on the order in which the random variables are observed.

3.2

$$\begin{aligned}\ell(\theta) &= log(L(\theta)) \\ &= log(\prod_{i=1}^{n}[\theta^{X_i}(1-\theta)^{1-X_i}]) \\ &= \sum_{i=1}^{n} log(\theta^{X_i}(1-\theta)^{1-X_i}) \\ &= \sum_{i=1}^{n}[X_i log(\theta) + (1-X_i)log(1-\theta)]\end{aligned}$$

$$\begin{aligned}\ell'(\theta) &= \frac{d}{d\theta}\sum_{i=1}^{n}[X_i log\theta + (1-X_i)log(1-\theta)] \\ &= \sum_{i=1}^{n}(\frac{X_i}{\theta} - \frac{1-X_i}{1-\theta}) \\ &= \frac{\sum_{i=1}^{n} X_i}{\theta} - \frac{\sum_{i=1}^{n}(1-X_i)}{1-\theta}\end{aligned}$$

If we set the above quantity to 0, we get the following series of equations:

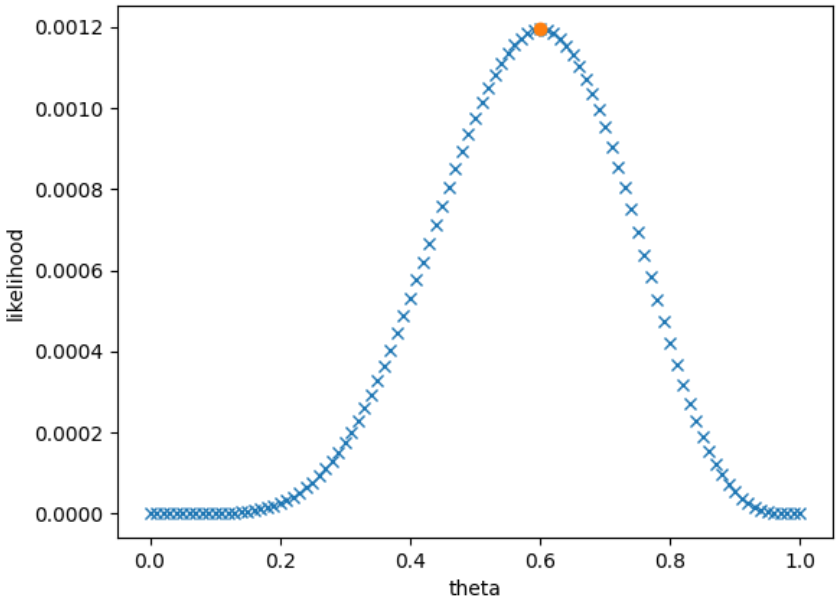
$$\begin{aligned}\frac{\sum_{i=1}^{n} X_i}{\theta} &= \frac{\sum_{i=1}^{n}(1-X_i)}{1-\theta} \\ (1-\theta)\sum_{i=1}^{n} X_i &= \theta\sum_{i=1}^{n}(1-X_i) \\ \sum_{i=1}^{n} X_i - \theta\sum_{i=1}^{n} X_i &= \theta n - \theta\sum_{i=1}^{n} X_i \\ \hat{\theta} &= \frac{\sum_{i=1}^{n} X_i}{n}\end{aligned}$$

Now let's find the second derivative of our log likelihood function to see if this function is convex. If it is convex, then there will only be one minimum, and this minimum will occur at the critical point $\hat{\theta}$, which we found above.

$$\ell''(\theta) = -\frac{\sum_{i=1}^{n} X_i}{\theta^2} - \frac{\sum_{i=1}^{n}(1-X_i)}{(1-\theta)^2}$$

We know that θ and X_i are nonnegative, so $\ell''(\theta) \leq 0$. Since our second derivative is always negative, we can conclude that the log likelihood function is convex and thus its only minimum will occur at the critical point we found above.

3.3

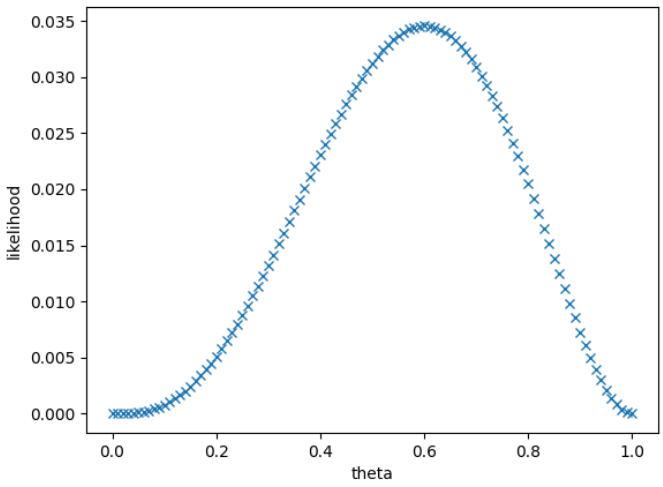


The value of $\hat{\theta}$ that maximizes the likelihood is 0.6. This answer does agree with our closed form answer.

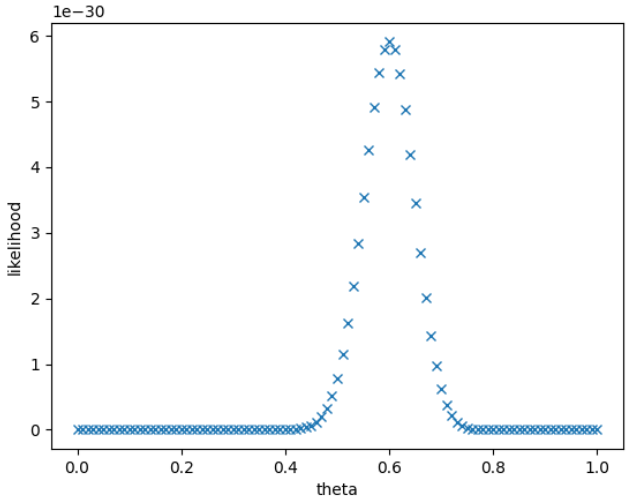
3.4

n	ones	MLE
10	6	0.6
5	3	0.6
100	60	0.6
10	5	0.5

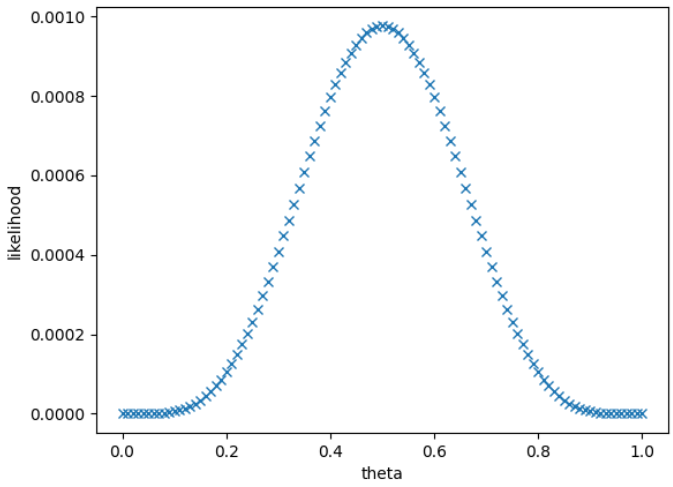
If n=5 and the number of ones is 3, our graph looks like this:



If n=100 and the number of ones is 60, our graph looks like this:



If n=10 and the number of ones is 5, our graph looks like this:



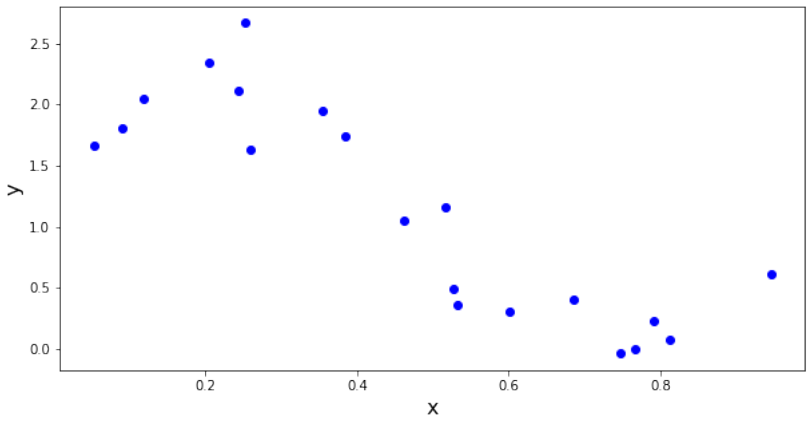
As seen in the table above, the MLEs are the same for entries that have the same proportion of ones. The actual likelihood values vary, but all have the shape of a bell-shaped curve. The more data points we have, the steeper the curve looks, which implies that our MLE has a higher likelihood. This is consistent with the idea that having more data points helps us better estimate theta.

4 Implementation: Polynomial Regression

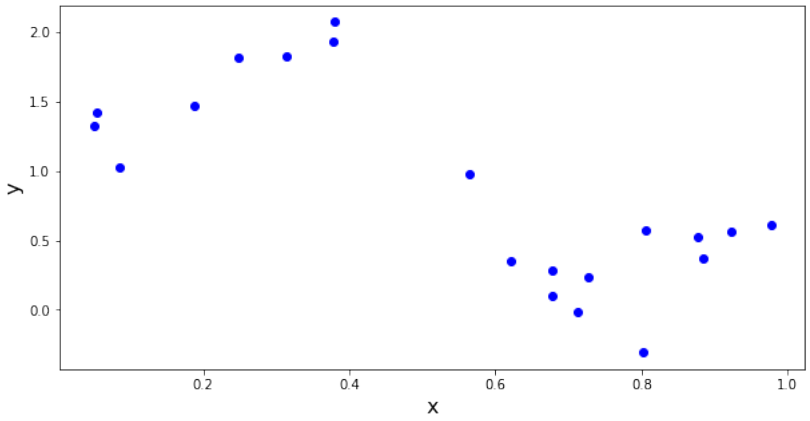
4.1 Visualization

```
# part a: main code for visualizations
print("Part A")
print('Visualizing data...')
print("Training data")
plot_data(train_data.X, train_data.y)
print("Testing data")
plot_data(test_data.X, test_data.y)
print("\n\n\n")
```

Below is the graph for our training data.



Below is the graph for our testing data.



I observe that the x and y scales are very similar for the two graphs. For the graph with the testing data, it looks like the data can be split into two clusters: one in the top left of the graph and another in the bottom right of the graph. For the top graph with the training data, it seems like the data does not follow a linear relationship very well as there are quite a few points that fall far from the projected line.

4.2

```
# part b: modify to create matrix for simple linear model
Phi = np.insert(X, 0, 1, axis=1)
```

4.3

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

4.4

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

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

η	iterations	final value	coefficients
10^{-6}	10000	25.8633	[0.3640 0.0922]
10^{-5}	10000	13.1589	[1.1570 -0.2252]
10^{-3}	7020	3.9126	[2.4464 -2.8164]
0.05	10000	nan	[nan nan]

With a small step size, the coefficients are both positive, but as we increase step size, the first coefficient is positive and the second is negative. The magnitude of the coefficients also increased as well as we increase step size, possibly because the larger the step size, the more likely we are to converge. The algorithm only converged for $\eta = 10^{-3}$. When I ran with 0.05, I obtained a runtime warning: overflow encountered in reduce. A possible explanation for this is that the step size was too large and our coefficients got too big.

4.5

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

The closed-form solution coefficients are [2.44640709 -2.81635359], and the cost is 3.9126. These are very similar to what we obtained in part d above for the iteration with $\eta = 10^{-3}$. We learned in class that the batch gradient descent algorithm takes $O(ND)$ runtime per iteration and that the closed-form solution takes $O(ND^2 + D^3)$ runtime. Here, since $N = 20$ and $D = 2$ and gradient descent algorithm requires 7020 iterations, our closed form solution will be faster in this case. However, if we were to increase N or D drastically, it's very likely that our gradient descent algorithm would perform better.

4.6

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

It takes 1731 iterations for the algorithm to converge with my proposed learning rate.

4.7

```
# part b: modify to create matrix for simple linear model
Phi = np.insert(X, 0, 1, axis=1)
# part g: modify to create matrix for polynomial model
m = self.m_
newPhi = []
for i in range(0, Phi.shape[0]):
  row = []
  for j in range(0, m + 1):
    row.append(pow(Phi[i][1], j))
  newPhi.append(row)
Phi = np.array(newPhi)
```

4.8

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

We might prefer RMSE as a metric over $J(\boldsymbol{\theta})$ because RMSE standardizes by dividing by the total number of data points we have. This is helpful when we compare RMSEs across datasets of different sizes.

4.9

```
print("Part I")
print('Investigating polynomial regression...')
m = range(11)
training_data_rms_errors = []
testing_data_rms_errors = []
for i in m:
  polyreg = PolynomialRegression(i)
  pr = polyreg.fit(train_data.X, train_data.y)
  training_data_rms_error = pr.rms_error(train_data.X, train_data.y)
  testing_data_rms_error = pr.rms_error(test_data.X, test_data.y)
  training_data_rms_errors.append(training_data_rms_error)
  testing_data_rms_errors.append(testing_data_rms_error)
  print("m = " + str(i))
  print("Training data rms error: " + str(training_data_rms_error))
```

```
  print("Testing data rms error: " + str(testing_data_rms_error))
plt.plot(m, training_data_rms_errors, label="training data rms error")
plt.plot(m, testing_data_rms_errors, label="testing data rms error")
plt.legend()
plt.xlabel("model complexity")
plt.ylabel("rms error")
```

I would say the 3rd degree polynomial fits this data best because although the 3rd, 4th, 5th, and 6th degree polynomials all have similar training data and test data rms errors, I assumed that a simpler model would be better. Since a 3rd degree polynomial is least complex, I picked that one. When $m = 8$, $m = 9$, and $m = 10$, there is evidence of overfitting because the training data rms error skyrocketed while the training data rms error stayed low. When $m = 0$, $m = 1$, $m = 2$, there is evidence of underfitting because the model had higher error rates. In fact, for $m = 0$, our model performed slightly better on the testing data than on the training data.

5 Code

5.1 Polynomial Regression Class

```
class PolynomialRegression() :

def __init__(self, m=1) :
    """
    Ordinary least squares regression.

    Attributes
```

```
        ---------------------
            coef_    -- numpy array of shape (d,)
                        estimated coefficients for the linear regression problem
            m_       -- integer
                        order for polynomial regression
        """
        self.coef_ = None
        self.m_ = m


    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
        Phi = np.insert(X, 0, 1, axis=1)
        # part g: modify to create matrix for polynomial model
        m = self.m_
        newPhi = []
        for i in range(0, Phi.shape[0]):
          row = []
          for j in range(0, m + 1):
            row.append(pow(Phi[i][1], j))
          newPhi.append(row)
        Phi = np.array(newPhi)

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

        return Phi


    def fit_GD(self, X, y, eta=None,
                 eps=0, tmax=10000, verbose=False) :
        """
```

```
    Finds the coefficients of a {d-1}^th degree polynomial
    that fits the data using least squares batch gradient descent.

    Parameters
    ----------
        X       -- numpy array of shape (n,d), features
        y       -- numpy array of shape (n,), targets
        eta     -- float, step size
        eps     -- float, convergence criterion
        tmax    -- integer, maximum number of iterations
        verbose -- boolean, for debugging purposes

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

    if verbose :
        plt.subplot(1, 2, 2)
        plt.xlabel('iteration')
        plt.ylabel(r'$J(\theta)$')
        plt.ion()
        plt.show()

    X = self.generate_polynomial_features(X) # map features
    n,d = X.shape
    eta_input = eta
    self.coef_ = np.zeros(d)                 # coefficients
    err_list  = np.zeros((tmax,1))           # errors per iteration

    # GD loop
    for t in range(tmax) :
        ### ========== 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 = float(1) / (1 + t) # change this line
        else :
            eta = eta_input
        ### ========== TODO : END ========== ###

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

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

            # stop?
            if t > 0 and abs(err_list[t] - err_list[t-1]) <= eps :
                break

            # debugging
            if verbose :
                x = np.reshape(X[:,1], (n,1))
                cost = self.cost(x,y)
                plt.subplot(1, 2, 1)
                plt.cla()
                plot_data(x, y)
                self.plot_regression()
                plt.subplot(1, 2, 2)
                plt.plot([t+1], [cost], 'bo')
                plt.suptitle('iteration: %d, cost: %f' % (t+1, cost))
                plt.draw()
                plt.pause(0.05) # pause for 0.05 sec

        print('number of iterations: %d' % (t+1))

        return self


    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
        self.coef_ = np.dot(np.linalg.pinv(np.dot(X.T, X)), np.dot(X.T, y))
        return self
        ### ========== TODO : END ========== ###


    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
        y = np.dot(self.coef_, X.T)
        ### ========== TODO : END ========== ###

        return y


    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 = sum(pow(self.predict(X)-y,2))
    ### ========== TODO : END ========== ###
    return cost


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
    error = np.sqrt(self.cost(X, y) / X.shape[0])
    ### ========== TODO : END ========== ###
    return error


def plot_regression(self, xmin=0, xmax=1, n=50, **kwargs) :
    """Plot regression line."""
    if 'color' not in kwargs :
        kwargs['color'] = 'r'
    if 'linestyle' not in kwargs :
        kwargs['linestyle'] = '-'

    X = np.reshape(np.linspace(0,1,n), (n,1))
    y = self.predict(X)
    plot_data(X, y, **kwargs)
    plt.show()
```

5.2 Main Function

```
#############################################################################
# main
#############################################################################
```

```
def main():
    # load data
    train_data = load_data(train_path)
    test_data = load_data(test_path)


    # ### ========== TODO : START ========== ###
    # part a: main code for visualizations
    print("Part A")
    print('Visualizing data...')
    print("Training data")
    plot_data(train_data.X, train_data.y)
    print("Testing data")
    plot_data(test_data.X, test_data.y)
    print("\n\n\n")
    # ### ========== TODO : END ========== ###



    # ### ========== TODO : START ========== ###
    # parts b-f: main code for linear regression
    # part b
    print("Part B")
    print('Investigating linear regression...')
    print("original")
    print(train_data.X)
    print("add ones column")
    polyreg = PolynomialRegression()
    X = polyreg.generate_polynomial_features(train_data.X)
    print(X)
    print("\n\n\n")

    # part d
    print("Part D")
    train_data = load_data(train_path)
    model = PolynomialRegression()
    model.coef_ = np.zeros(2)
    print("Model cost should equal 40.234. Our code results in model cost of " + str(model.c

    etas = [pow(10, -6), pow(10, -5), pow(10, -3), 0.05]
    for e in etas:
      print("For eta = " + str(e) + ",")
      m = model.fit_GD(train_data.X, train_data.y, eta=e)
      print("cost: " + str(m.cost(train_data.X, train_data.y)))
      print("coefficients: " + str(m.coef_) + "\n")
```

```
print("\n\n")

# part e
print("Part E")
m = model.fit(train_data.X, train_data.y)
print("closed form coefficients: " + str(m.coef_))
print("closed form cost: " + str(m.cost(train_data.X, train_data.y)) + "\n\n\n")

# part f
print("Part F")
print("After setting a learning rate for GD that is a function of k,")
m = model.fit_GD(train_data.X, train_data.y)
print(m.coef_)
print("\n\n\n")

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



### ========== TODO : START ========== ###
# parts g-i: main code for polynomial regression
print("Part I")
print('Investigating polynomial regression...')
m = range(11)
training_data_rms_errors = []
testing_data_rms_errors = []
for i in m:
  polyreg = PolynomialRegression(i)
  pr = polyreg.fit(train_data.X, train_data.y)
  training_data_rms_error = pr.rms_error(train_data.X, train_data.y)
  testing_data_rms_error = pr.rms_error(test_data.X, test_data.y)
  training_data_rms_errors.append(training_data_rms_error)
  testing_data_rms_errors.append(testing_data_rms_error)
  print("m = " + str(i))
  print("Training data rms error: " + str(training_data_rms_error))
  print("Testing data rms error: " + str(testing_data_rms_error))
plt.plot(m, training_data_rms_errors, label="training data rms error")
plt.plot(m, testing_data_rms_errors, label="testing data rms error")
plt.legend()
plt.xlabel("model complexity")
plt.ylabel("rms error")
### ========== TODO : END ========== ###


print("Done!")
```

```python
if __name__ == "__main__":
    main()
```