CM146, Fall 2020
Problem Set 2: Linear Classification and Logistic Regression
Due Nov. 12 at 11:59 pm

Ian Conceicao, 505153981

1 Linear Model [15 pts] Thursday, Nov 12

(a) Design a two-input linear model $w_1X_1 + w_2X_2 + b$ that computes the following Boolean functions. Assume $T = 1$ and $F = 0$. If a valid linear model exists, show that it is not unique by designing another valid linear model. If no such linear model exists, explain why.

(i) OR [5 pts]

(ii) XOR [5 pts]

Solution:
(i) Let's create 2 different set of weights that follow the rules:
if $w_1X_1 + w_2X_2 + b < 0$ then F
if $w_1X_1 + w_2X_2 + b \geq 0$ then T
Two possible weight configurations are:
$w_1 = 1, w_2 = 1, b = 0$
$w_1 = 10, w_2 = 10, b = -5$
(ii) No such linear model exists to compute an XOR function. The XOR target space is not linearly separable, so it cannot be modeled linearly

(b) How many distinct Boolean functions are possible with two Boolean variables? Out of them, how many can be represented by a linear model? Justify your answers. [5 pts]

Solution: There are 16 possible Boolean functions that are possible with two Boolean variables. We can find this value by recognizing there are 2 possible values for x_1, x_2. That leaves an input space of 4. We can create 2^4 different functions by choosing 2^4 different combinations of output spaces. Thus we have 16 possible functions. 14 of these functions can be modeled linearly. I reached this conclusion because in the target space there are 6 possible ways to separate the data, and for each line, either side can be switched to mean positive or negative. In other words if you draw line f(x), y > f(x) can mean 1 or 0. This leaves us with 12 functions, and once add on the all 0 output and all 1 output we have 14 functions.

2 Logistic Regression and its Variant [45 pts]

Consider the logistic regression model for binary classification that takes input features $\boldsymbol{x}_n \in R^m$ and predicts $y_n \in \{1, 0\}$. As we learned in class, the logistic regression model fits the probability

Parts of this assignment are adapted from course material by Andrew Ng (Stanford), Jenna Wiens (UMich) and Jessica Wu (Harvey Mudd).

$P(y_n = 1)$ using the *sigmoid* function:

$$P(y_n = 1) = h(\boldsymbol{x}_n) = \sigma(\boldsymbol{w}^T \boldsymbol{x}_n + b) = \frac{1}{1 + \exp(-\boldsymbol{w}^T \boldsymbol{x}_n - b)}. \tag{1}$$

Given N training data points, we learn the logistic regression model by minimizing the negative log-likelihood:

$$J(\boldsymbol{w}, b) = -\sum_{n=1}^{N} [y_n \log h(\boldsymbol{x}_n) + (1 - y_n) \log (1 - h(\boldsymbol{x}_n))]. \tag{2}$$

(a) In the following, we derive the stochastic gradient descent algorithm for logistic regression. [20 pts]

i. Partial derivatives $\frac{\partial J}{\partial w_j}$ and $\frac{\partial J}{\partial b}$, where w_j is the j-th element of the weight vector $\boldsymbol{w}$. [5 + 5 pts]

Solution:

$$\begin{aligned}
&\tfrac{\partial}{\partial w_j} - \textstyle\sum_{n=1}^{N} [y_n \log h(x_n) + (1 - y_n) \log (1 - h(x_n))] \\
&- \textstyle\sum_{n=1}^{N} \tfrac{\partial}{\partial w_j} [y_n \log h(x_n) + (1 - y_n) \log (1 - h(x_n))] \\
&= - \textstyle\sum_{n=1}^{N} \left[\tfrac{\partial}{\partial w_j} y_n \log h(x_n) + \tfrac{\partial}{\partial w_j} (1 - y_n) \log (1 - h(x_n)) \right] \\
&= - \textstyle\sum_{n=1}^{N} \left[\tfrac{y_n}{h(x_n)} + \tfrac{1-y_n}{1-h(x_n)} \right] \tfrac{\partial}{\partial w_j} h(x_n) \\
&= - \textstyle\sum_{n=1}^{N} \left[\tfrac{y_n}{h(x_n)} + \tfrac{1-y_n}{1-h(x_n)} \right] h(x_n) (1 - h(x_n)) x_{n,j} \\
&= - \textstyle\sum_{n=1}^{N} \left[\tfrac{y_n - h(x_n)}{h(x_n)(1-h(x_n))} \right] h(x_n) (1 - h(x_n)) x_{n,j} \\
&\boxed{= - \sum_{n=1}^{N} [y_n - h(x_n)] x_{n,j}}
\end{aligned}$$

Taking the derivative of the negative log-likelihood function with respect to b is a special case of $w_j = b$. We know that for any b, the associated $x_{n,j}$, for every n is 1. In other words, if we treat $b = w_0$, then any $x_{n,0} = 1$. Therefore our $\frac{\partial J}{\partial b}$ is:

$$\boxed{= - \sum_{n=1}^{N} y_n - h(x_n)}$$

ii. Write down the stochastic gradient descent algorithm for minimizing Eq. (2) using the partial derivatives computed above. [5 pts]

Solution: Repeat until converge:

i. Randomly pick one sample (x_i, y_i)
ii. For every w_j:
    A. $w_j = w_j - \eta [y_i - h(x_i)] x_{i,j}$
iii. $b = b - \eta [y_i - h(x_i)]$

iii. Compare the stochastic gradient descent algorithm for logistic regression with the Perceptron algorithm. What are the similarities and what are the differences? [5 pts]

Solution: Both the stochastic gradient descent algorithm and the Perceptron algorithm process one data point at a time. Both algorithms also utilize the Convergence

Theorem: If there exist a set of weights that are consistent with the data (ie.e the data is linearly separable) the algorithm will converge. However they both suffer from the cycling theorem: the training data is not linearly separable, then the learning algorithm will eventually repeat the same set of weights and enter an infinite loop. However the logistic regression learning algorithm utilizes derivatives to update weights, while the Perceptron algorithm uses only the output and input. The update step is: $w = w + yx$

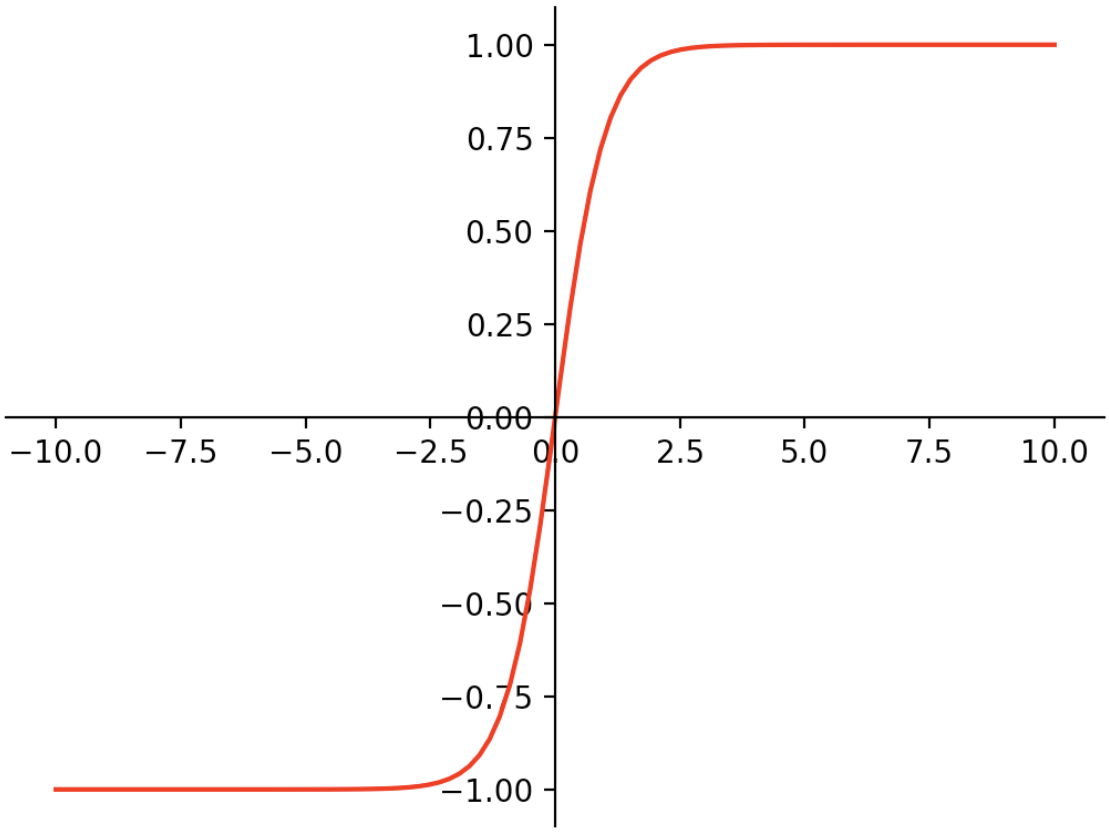
(b) Instead of using the *sigmoid* function, we would like to use the following transformation function:

$$\sigma_A(z) = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}.$$

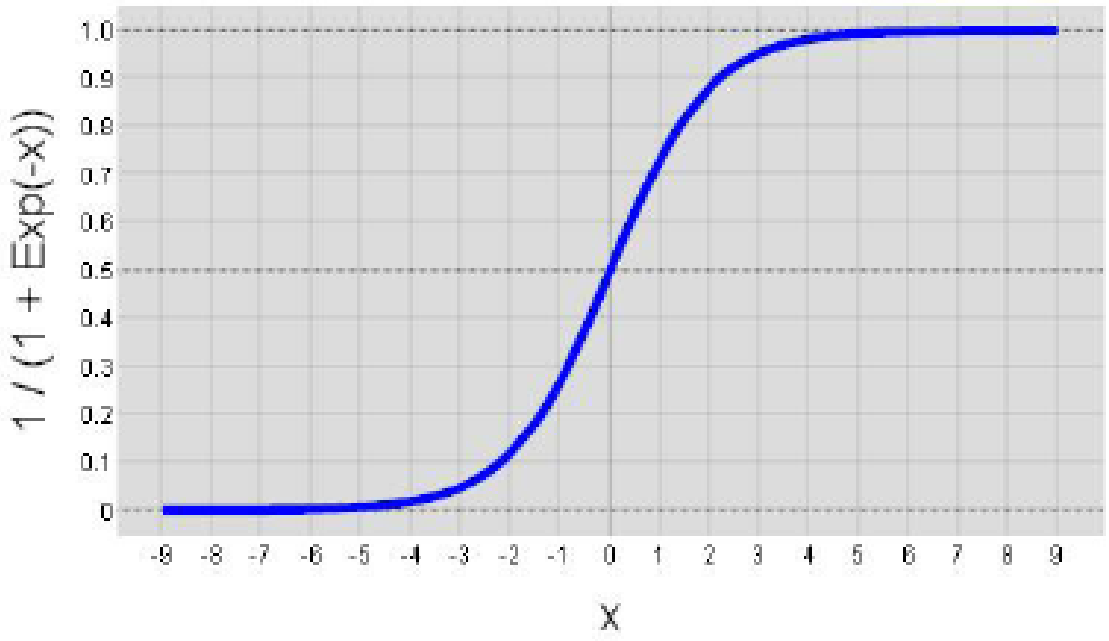
Answer the following questions [25 pts]:

i. Plot $\sigma_A(z)$ as a function of z in python using *matplotlib* and *numpy* libraries. Consider $z \in [-10, 10]$ for the plot. What are the similarities and differences between σ and σ_A? What happens as $z \to \infty$ and $z \to -\infty$. [5 pts]

Solution:



Above is a plot of σ_A. I created this plot using numpy and matplotlib.



Above is a plot of sigmoid. I found it on: https://www.researchgate.net/figure/Sigmoid-squashing-function-1-1-Exp-x_fig1_228359669
The range of σ_A is $(-1, 1)$ while the range of σ is $(0, 1)$. Both have an inflection point at x = 0, with σ_A equaling 0 at $x = 0$ and σ equaling 0.5 at $x = 0.0$. Both approach 1 as x approaches infinity, however σ_A approaches -1 as x approaches negative infinity and σ approaches 0 as x approaches negative infinity.

ii. Prove the following: [5 pts]

$$\frac{d\sigma_A(z)}{dz} = 1 - \sigma_A^2(z).$$

Solution: For intuitiveness, let us use e^x instead of $\exp(x)$.

$$\begin{aligned}
&\frac{d\sigma_A(z)}{dz} \\
&= \frac{d}{dz}\frac{e^z - e^{-z}}{e^z + e^{-z}} \\
&= \frac{(\frac{d}{dz}(e^z - e^{-z}))(e^z + e^{-z}) - (\frac{d}{dz}(e^z + e^{-z}))(e^z - e^{-z})}{(e^z + e^{-z})^2} \\
&= \frac{(e^z + e^{-z})(e^z + e^{-z}) - (e^z - e^{-z})(e^z - e^{-z})}{(e^z + e^{-z})(e^z + e^{-z})} \\
&= \frac{(e^{2z} + e^0 + e^0 + e^{-2z}) - (e^{2z} - e^0 - e^0 + e^{-2z})}{(e^{2z} + e^0 + e^0 + e^{-2z})} \\
&= 1 - \frac{e^{2z} - e^0 - e^0 + e^{-2z}}{e^{2z} + e^0 + e^0 + e^{-2z}} \\
&= 1 - \frac{(e^x - e^{-x})^2}{(e^x + e^{-x})^2} \\
&= 1 - (\frac{e^x - e^{-x}}{e^x + e^{-x}})^2 \\
&= 1 - \sigma_A^2(z)
\end{aligned}$$

iii. Can we assume probability $P(y_n = 1) = \sigma_A(\boldsymbol{w}^T \boldsymbol{x}_n + b)$? Why or why not? [2 pts]

Solution: No, we cannot make that assumption. We cannot have a negative probability, and the range of the σ_A is (-1,1).

iv. If we assume

$$P(y_n = 1) = h_A(\boldsymbol{x}_n) = \frac{1 + \sigma_A(\boldsymbol{w}^T \boldsymbol{x}_n + b)}{2}.$$

Given N examples, $\{\boldsymbol{x}_n, y_n\}_{n=1}^N$, please write down the corresponding negative log-likelihood function $J_A(\boldsymbol{w}, b)$. [3 pts]

Solution:

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

$l(\theta) = \sum_{n=1}^N y_n log(h_A(x_n)) + (1 - y_n)log(1 - h_A(x_n))$

$J_A(w, b) = -\sum_{n=1}^N y_n log(h_A(x_n)) + (1 - y_n)log(1 - h_A(x_n))$

$J_A(w, b) = -\sum_{n=1}^N y_n log(\frac{1+\sigma_A(w^T x_n+b)}{2}) + (1 - y_n)log(1 - \frac{1+\sigma_A(w^T x_n+b)}{2})$

$$\boxed{J_A(w, b) = -\sum_{n=1}^N y_n log(\frac{1 + \sigma_A(w^T x_n + b)}{2}) + (1 - y_n)log(\frac{1 - \sigma_A(w^T x_n + b)}{2})}$$

v. Compute the partial derivatives $\frac{\partial J_A}{\partial w_j}$ and $\frac{\partial J_A}{\partial b}$. [5 pts]

Solution: $\frac{\partial J_A}{\partial w_j}$

$= \frac{\partial}{\partial w_j} - \sum_{n=1}^N y_n log(\frac{1+\sigma_A(w^T x_n+b)}{2}) + (1 - y_n)log(\frac{1-\sigma_A(w^T x_n+b)}{2})$

$= -\sum_{n=1}^N y_n \frac{\partial}{\partial w_j} log(\frac{1+\sigma_A(w^T x_n+b)}{2}) + (1 - y_n)\frac{\partial}{\partial w_j} log(\frac{1-\sigma_A(w^T x_n+b)}{2})$

$= -\sum_{n=1}^N y_n(\frac{2}{1+\sigma_A(w^T x_n+b)})\frac{\partial}{\partial w_j}(\frac{1+\sigma_A(w^T x_n+b)}{2}) + (1-y_n)(\frac{2}{1-\sigma_A(w^T x_n+b)})\frac{\partial}{\partial w_j}(\frac{1-\sigma_A(w^T x_n+b)}{2})$

$= -\sum_{n=1}^N y_n(\frac{2}{1+\sigma_A(w^T x_n+b)})(\frac{1}{2})(1-\sigma_A^2(w^T x_n+b))\frac{\partial}{\partial w_j}(w^T x_n+b) + (1-y_n)(\frac{2}{1-\sigma_A(w^T x_n+b)})(\frac{-1}{2})(1-\sigma_A^2(w^T x_n + b))\frac{\partial}{\partial w_j}((w^T x_n + b))$

$= -\sum_{n=1}^N \frac{y_n(1-\sigma_A^2(w^T x_n+b))x_{n,j}}{1+\sigma_A(w^T x_n+b)} - \frac{(1-y_n)(1-\sigma_A^2(w^T x_n+b))x_{n,j}}{1-\sigma_A(w^T x_n+b)}$

$= -\sum_{n=1}^N (1 - \sigma_A^2(w^T x_n + b))x_{n,j}[\frac{y_n}{1+\sigma_A(w^T x_n+b)} - \frac{(1-y_n)}{1-\sigma_A(w^T x_n+b)}]$

$= -\sum_{n=1}^N (1 - \sigma_A^2(w^T x_n + b))x_{n,j}[\frac{y_n(1-\sigma_A(w^T x_n+b)) - (1-y_n)(1+\sigma_A(w^T x_n+b)}{1-\sigma_A^2(w^T x_n+b)}]$

$= -\sum_{n=1}^N x_{n,j}[y_n(1 - \sigma_A(w^T x_n + b)) - (1 - y_n)(1 + \sigma_A(w^T x_n + b)]$

$= -\sum_{n=1}^N x_{n,j}[y_n(1 - \sigma_A(w^T x_n + b)) - (1 + \sigma_A(w^T x_n + b) + y_n(1 + \sigma_A(w^T x_n + b)]$

$$\boxed{\frac{\partial J_A}{\partial w_j} = -\sum_{n=1}^N [2y_n - \sigma_A(w^T x_n + b) - 1]x_{n,j}}$$

Because b is a special case where all $X_{n,b} = 1$,

$$\boxed{\frac{\partial J_A}{\partial w_b} = -\sum_{n=1}^N 2y_n - \sigma_A(w^T x_n + b) - 1}$$

vi. Write down the stochastic gradient descent algorithm for minimizing $J_A(\boldsymbol{w}, b)$. [5 pts]

Solution: Repeat until converge:

i. Randomly pick one sample (x_i, y_i)

ii. For every w_j:

   A. $w_j = w_j - \eta[2y_i - \sigma_A(w^T x_i + b) - 1]x_{i,j}$

iii. $b = b - \eta[2y_n - \sigma_A(w^T x_n + b) - 1]$

3 Implementation: Polynomial Regression [40 pts]

In this exercise, you will work through linear and polynomial regression. Our data consists of inputs $x_n \in \mathbb{R}$ and outputs $y_n \in \mathbb{R}, n \in \{1, \ldots, N\}$, which are related through a target function $y = f(x)$. Your goal is to learn a linear predictor $h_{\boldsymbol{w}}(x)$ that best approximates $f(x)$. But this time, rather than using `scikit-learn`, we will further open the "black-box", and you will implement the regression model!

---

code and data

- code : `Fall2020-CS146-HW2.ipynb`
- data : `train.csv`, `test.csv`

Please use your *@g.ucla.edu* email id to access the code and data. Similar to *HW-1*, copy the colab notebook to your drive and make the changes. Mount the drive appropriately and copy the shared data folder to your drive to access via colab. For colab usage demo, check out the Discussion recordings for Week 2 in CCLE. The notebook has marked blocks where you need to code.

========= TODO : START =========

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

Note: For the questions requiring you to complete a piece of code, you are expected to copy-paste your code as a part of the solution in the submission pdf. Tip: If you are using LaTeX, check out the Minted package (example) for code highlighting.

This is likely the first time that many of you are working with `numpy` and matrix operations within a programming environment. For the uninitiated, you may find it useful to work through a `numpy` tutorial first.[1] Here are some things to keep in mind as you complete this problem:

- If you are seeing many errors at runtime, inspect your matrix operations to make sure that you are adding and multiplying matrices of compatible dimensions. Printing the dimensions of variables with the `X.shape` command will help you debug.
- When working with `numpy` arrays, remember that `numpy` interprets the `*` operator as element-wise multiplication. This is a common source of size incompatibility errors. If you want matrix multiplication, you need to use the `dot` function in Python. For example, `A*B` does element-wise multiplication while `dot(A,B)` does a matrix multiply.
- Be careful when handling `numpy` vectors (rank-1 arrays): the vector shapes $1 \times N$, $N \times 1$, and N are all different things. For these dimensions, we follow the the conventions of `scikit-learn`'s `LinearRegression` class[2]. Most importantly, unless otherwise indicated (in the code documentation), both column and row vectors are rank-1 arrays of shape N, not rank-2 arrays of shape $N \times 1$ or shape $1 \times N$.

Visualization [2 pts]

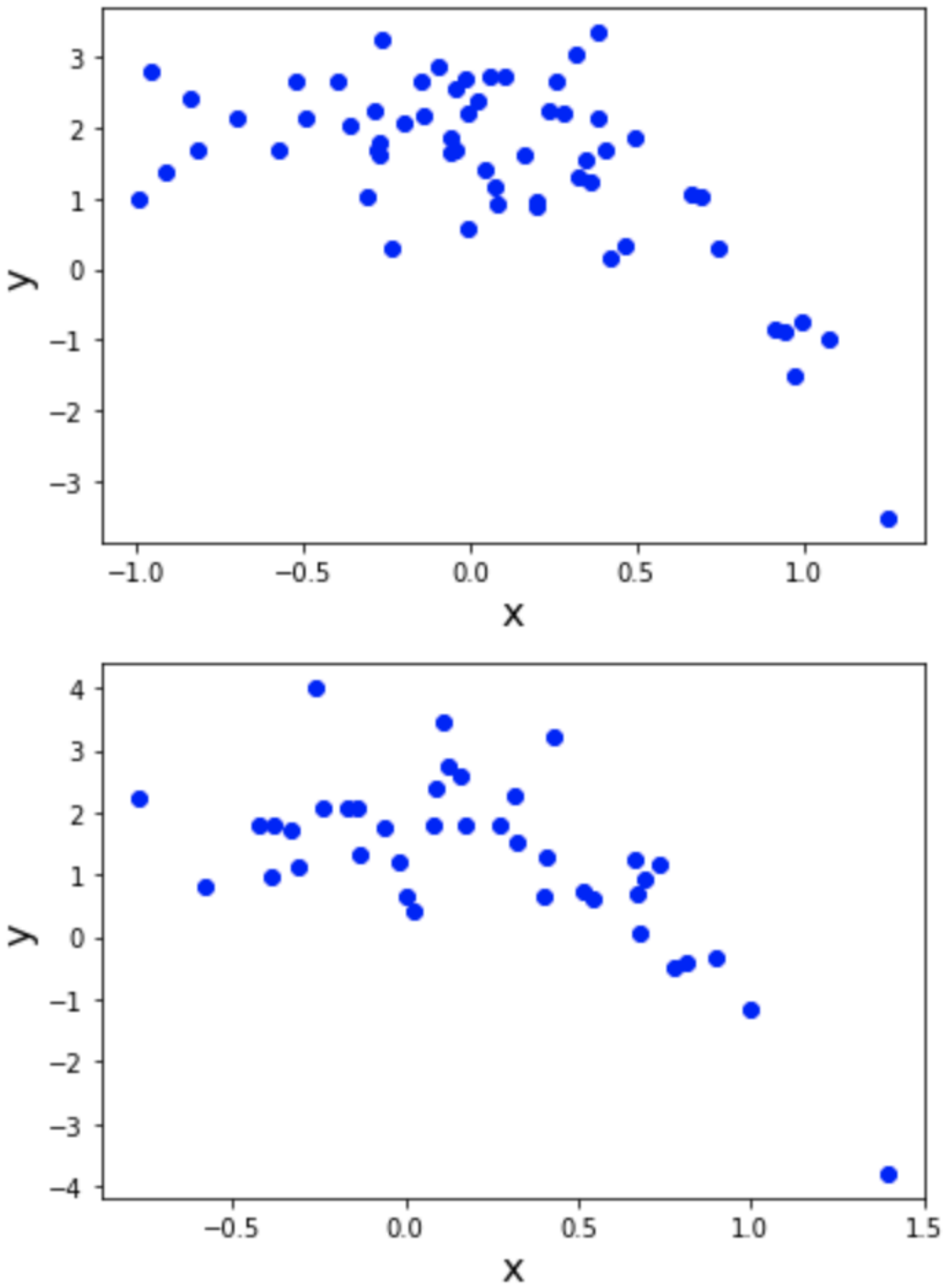
[1] Try out `SciPy`'s tutorial (http://wiki.scipy.org/Tentative_NumPy_Tutorial), or use your favorite search engine to find an alternative. Those familiar with Matlab may find the "Numpy for Matlab Users" documentation (http://wiki.scipy.org/NumPy_for_Matlab_Users) more helpful.

[2] http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html

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) 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]

Solution: The figures became:



The top figure is the training data and the bottom figure is the test data. Both data appears

to have noise. Finding a perfect fit will not be possible. For example, on the test data look at x = 0. Close to x=0 we have many data points that appear to be at y=0, and many data points that are far closer to y=3. The same goes for the test data. Both datasets also do not appear linear. The slope seems to start off close to 0 then decay quickly closer to -4 or so just from a casual glance. Because the data does not have a constant slope, I do not think linear regression will be effective at predicting the data. Below is the code:
First I gave the path to the data:

```
data_directory_path = '/content/drive/My Drive/01/Classes/Junior Q1/CS 146/Homeworks/HW2'
```

Then I plotted the figures:

```
plot_data(train_data.X, train_data.y)
plot_data(test_data.X, test_data.y)
```

Linear Regression [23 pts]

Recall that linear regression attempts to minimize the objective function

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

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

$$\boldsymbol{y} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix}, \qquad \boldsymbol{X} = \begin{pmatrix} \boldsymbol{x}_1^T \\ \boldsymbol{x}_2^T \\ \vdots \\ \boldsymbol{x}_N^T \end{pmatrix}, \qquad \boldsymbol{w} = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \\ \vdots \\ w_D \end{pmatrix}$$

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

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:

$$h_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{x} = w_0 + w_1 x_1$$

`regression.py` 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, $\big(1, x_{n,1}, x_{n,1}^2, \ldots, x_{n,1}^m\big)^T$. Setting $m = 1$ instantiates an object where the feature vector for instance n, $\big(1, x_{n,1}\big)^T$.

(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.

Solution:

```
Phi = np.ones_like(X) #Make first column of 1's
m = self.m_
Phi = Phi.hstack((Phi,X)) #Combine 2 columns
```

(c) Before tackling the harder problem of training the regression model, complete `PolynomialRegression.predict(...)` to predict $\boldsymbol{y}$ from $\boldsymbol{X}$ and $\boldsymbol{w}$. [3 pts]

Solution:

```
y = np.dot(X, self.coef_)
```

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

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

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

In gradient descent, each iteration performs the update

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

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

- **(2 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 $J(\boldsymbol{w})$.

  If you have implemented everything correctly, then the following code snippet should print the model cost as 230.867214.

  ```
  model = PolynomialRegression(1)
  model.coef_ = np.zeros(2)
  c = model.cost (train_data.X, train_data.y)
  print(f'model_cost:{c}')
  ```

  Solution:

  ```
  y_hat = self.predict(X)
  mean_squares = (y - y_hat) ** 2
  cost = sum(mean_squares)
  ```

- **(3 pts)** 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 $\boldsymbol{w}$ and the new predictions $\hat{y} = h_{\boldsymbol{w}}(\boldsymbol{x})$ 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.

Solution:

```
error = self.predict(X) - y
gradient = np.dot(X.T, error)
self.coef_ = self.coef_ - (2 * eta * gradient)

# 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)
```

- **(5 pts)** Experiment with different values of learning rate $\eta = 10^{-6}$, 10^{-5}, 10^{-3}, 0.0168 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 η = 0.0168? Explain the observation and causes.

  Solution: Below is the table. Below that was the code to output the information. The 2 lowest cost functions, the 2nd and 3rd entries in the table, have the most similar coefficients. Along with their low costs, it shows these 2 learning rates allowed the model to get closest to a good fit. Because the 3rd learning rate, 10^-3, only needed 503 iterations before passing the epsilon threshold, and has the lowest cost, it is the best learning rate for the data. The 4th learning rate, 0.0168 is far too high, and not only did it never converge, but the cost function is 223 order of magnitudes higher than the other cost functions, and the coefficients are 110 powers higher than all other coefficients for the other learning rates. This implies the learning rate was too large, and the weights were adjusted too greatly with each pass. The third algorithm was by far the quickest, and the other 3 had marginally different runtimes. To acquire the runtimes I had to import the *time* library of python, so the code will not work if that module is not imported.

Learning Rate	Coefficients	Iterations	Time Elapsed (seconds)	Cost
10^{-6}	1.059, -0.352	10,000	0.3173	93.768
10^{-5}	1.586,-1.417	10,000	0.3294	60.483
10^{-3}	1.591 -1.484	503	0.0206	60.41
0.0168	$-1.305 * 10^{111}$,$-1.065 * 10^{110}$	10,000	0.3026	$1.033 * 10^{224}$

```
print("-----")
print("Learning Rate = {}".format(eta))
print("Iterations = {}".format(t+1))
print("Cost = {}".format(self.cost(X,y)))
print("Coefficients:")
```

```
print(self.coef_)
print("Time {}".format(str(end-begin)))
```

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

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

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

- **(2 pts)** Implement the closed-form solution `PolynomialRegression.fit(...)`.
  Solution:

```
        begin = time()
        self.coef_ = np.linalg.pinv((np.dot(X.T, X))).dot(X.T).dot(y)
        end = time()

        print("-----")
        print("Cost = {}".format(self.cost(X,y)))
        print("Coefficients:")
        print(self.coef_)
        print("Time {}".format(str(end-begin)))
```

- **(2 pts)** 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?

  Solution: The closed-form solution coefficents are:
  1.591, -1.484
  These are the same coefficients obtained by the one trial of FD that converged. The cost of the closed-form solution is:
  60.41
  This is the same cost as the GD algorithm mentioned. The closed form solution ran in 0.00086 seconds, which was far below the best GD time of 0.0206 seconds.

(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]

Solution: The algorithm converged with only 357 iterations. This is much better than the previous best convergence of 503 iterations with a fixed learning rate.

Polynomial Regression[15 pts]

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

$$h_{\boldsymbol{w}}(\boldsymbol{x}) = \boldsymbol{w}^T\phi(\boldsymbol{x}) = w_0 + w_1x + w_2x^2 + \ldots + w_mx^m.$$

(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. [4 pts]

Solution:

```
        m = self.m_
        #Check if the data is already good:
        if np.shape(X)[1] == m + 1:
          return X
        Phi = np.ones_like(X) #Make first column of 1's

        for i in range(0, m):
            j = i+1
            Phi = np.hstack((Phi, X**j))
```

(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

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

where N is the number of instances.[3]

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

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

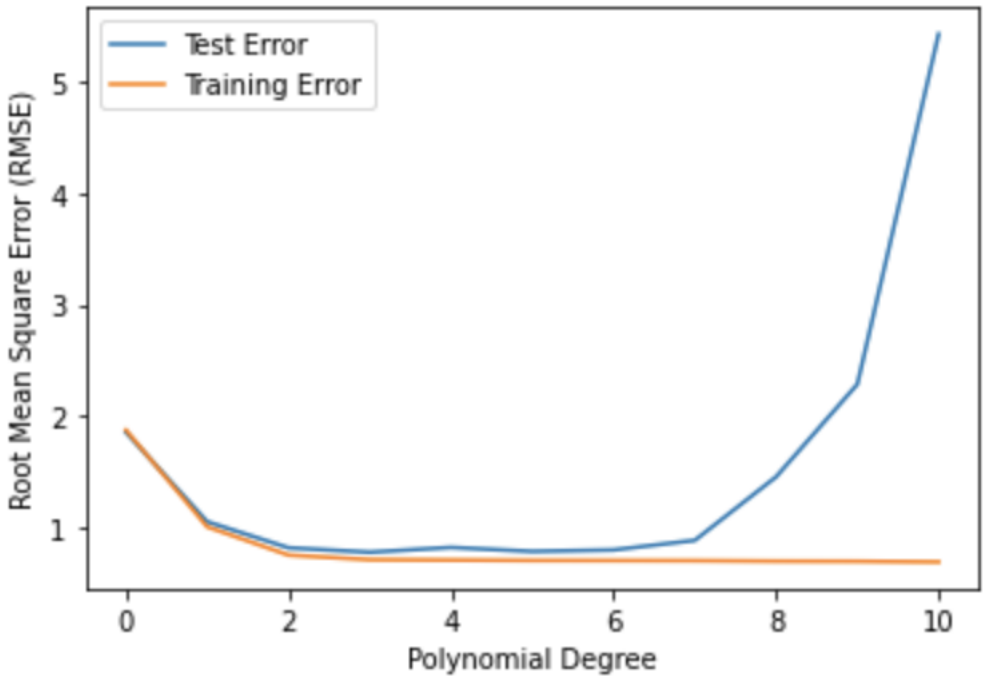
Solution: RMSE normalizes the error for different N number of data points. Also, because RMSE square roots the MSE, the result is the standard deviation of the prediction errors. Knowing the standard deviation of residuals tells us how concentrated the data is around our best fit function. We would use this error evaluation over $J(\theta)$ because $J(\theta)$ is not normalized for the number of training data. Any non-perfect fit to the data would result in an increase in the error when using $J(\theta)$. Meanwhile adding a data point that is more accurate than the others in the data set while using RMSE would decrease the RMSE, to reflect the function better fits the data.

```
J = self.cost(self.generate_polynomial_features(X), y)
error = math.sqrt(J/X.shape[0])
```

[3] Note that the RMSE as defined is a biased estimator. To obtain an unbiased estimator, we would have to divide by $n-k$, where k is the number of parameters fitted (including the constant), so here, $k = m+1$.

(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 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. [7 pts]

Solution: Below is the graph of the Polynomial Degree fitting and the Root Mean Square Errors associate with each model. We observe that while training error continues to decrease as the polynomial order increases, test error does not follow. In fact test error becomes far worse after more than 6 degrees. This is due to data overfitting. The model begins to fit the training data too accurately, and when it encounters new data it cannot generalize to the new data. The best order is the order that performs best on the test data, which I found to be order 3, with a test RMSE score of 0.775.




```
test_errors = []
train_errors = []
orders = [*range(0,11)]

bestError = None
bestM = None

for orderM in orders:
  print("Ran")
  model = PolynomialRegression(orderM)
  model.fit(train_data.X, train_data.y)
  RMSE_train = model.rms_error(train_data.X, train_data.y)
  RMSE_test = model.rms_error(test_data.X, test_data.y)
```

```
    train_errors.append(RMSE_train)
    test_errors.append(RMSE_test)

    if bestError == None or RMSE_test < bestError:
      bestError = RMSE_test
      bestM = orderM

  plt.figure()
  plt.plot(orders, test_errors, label="Test Error")
  plt.plot(orders, train_errors, label="Training Error")
  plt.legend()
  plt.xlabel("Polynomial Degree")
  plt.ylabel("Root Mean Square Error (RMSE)")
  plt.show()

  print("Best order polynomial was: {} with an error of {}".format(bestM,bestError))
```