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

1 Linear Model [15 pts]

(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)$20X_1 + 20X_2 - 10 \quad 30X_1 + 20X_2 - 10$
(ii)No such linear model exists because XOR is not linearly separable. We can easily see from the graph that we are trying to separate the two set of points {(0,0), (1,1)} and {(1,0), (0,1)}. They are clearly not linearly separable.

(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 4 different kinds of input values and for each input we have two different output, so there are $\boxed{2^4 = 16}$ distinct Boolean functions. $\boxed{14}$ of them can be represented by a linear model. The only two that are not linearly separable are XOR and XNOR. We can easily see from the graph that we are trying to separate the two set of points on the diagonals of a square. They are clearly not linearly separable.

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 $P(y_n = 1)$ 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:

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

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

(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:

$$h(\boldsymbol{x}_n) = \sigma(\boldsymbol{w}^T\boldsymbol{x}_n + b) = \sigma(w_1 x_1^{(n)} + ... + w_m x_m^{(n)} + b)$$

$$\frac{\partial h(\boldsymbol{x}_n)}{\partial w_j} = \frac{\partial \sigma(\boldsymbol{w}^T\boldsymbol{x}_n + b)}{\partial(\boldsymbol{w}^T\boldsymbol{x}_n + b)}\frac{\partial(\boldsymbol{w}^T\boldsymbol{x}_n + b)}{\partial w_j} = \sigma(\boldsymbol{w}^T\boldsymbol{x}_n+b)(1-\sigma(\boldsymbol{w}^T\boldsymbol{x}_n+b))x_j^n = h(\boldsymbol{x}_n)(1-h(\boldsymbol{x}_n))x_j^n$$

$$\frac{\partial J}{\partial w_j} = \frac{\partial J}{\partial h(\boldsymbol{x}_n)}\frac{\partial h(\boldsymbol{x}_n)}{\partial w_j} = -\sum_{n=1}^{N}[\frac{y_n}{h(\boldsymbol{x}_n)} - \frac{(1-y_n)}{(1-h(\boldsymbol{x}_n))}]h(\boldsymbol{x}_n)(1-h(\boldsymbol{x}_n))x_j^n$$

$$= -\sum_{n=1}^{N}[y_n(1-h(\boldsymbol{x}_n))x_j^n - (1-y_n)h(\boldsymbol{x}_n)x_j^n] = \boxed{\sum_{n=1}^{N}[h(\boldsymbol{x}_n)x_j^n - y_n x_j^n]}$$

$$\frac{\partial h(\boldsymbol{x}_n)}{\partial b} = \frac{\partial \sigma(\boldsymbol{w}^T\boldsymbol{x}_n + b)}{\partial(\boldsymbol{w}^T\boldsymbol{x}_n + b)}\frac{\partial(\boldsymbol{w}^T\boldsymbol{x}_n + b)}{\partial b} = \sigma(\boldsymbol{w}^T\boldsymbol{x}_n+b)(1-\sigma(\boldsymbol{w}^T\boldsymbol{x}_n+b)) = h(\boldsymbol{x}_n)(1-h(\boldsymbol{x}_n))$$

$$\frac{\partial J}{\partial b} = \frac{\partial J}{\partial h(\boldsymbol{x}_n)}\frac{\partial h(\boldsymbol{x}_n)}{\partial b} = -\sum_{n=1}^{N}[\frac{y_n}{h(\boldsymbol{x}_n)} - \frac{(1-y_n)}{(1-h(\boldsymbol{x}_n))}]h(\boldsymbol{x}_n)(1-h(\boldsymbol{x}_n))$$

$$= -\sum_{n=1}^{N}[y_n(1-h(\boldsymbol{x}_n)) - (1-y_n)h(\boldsymbol{x}_n)] = \boxed{\sum_{n=1}^{N}[h(\boldsymbol{x}_n) - y_n]}$$

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

Solution:

```
1. Initialize $\boldsymbol{w}$. Initialize the learning rate r
2. For t = 0, 1, 2, .... (until error below some threshold)
      For each training example $(\boldsymbol{x}^i, y^i)$:
          $b^{t+1} = b^t - r(h(\boldsymbol{x}^i) - y^i)$
          $b^t = b^{t+1}$
          For each element of the weight vector $(w_j)$:
              $w_j^{t+1} = w_j^t - r(h(\boldsymbol{x}^i)x_j^i - y^i x_j^i)$
              $w_j^t = w_j^{t+1}$
return $\boldsymbol{w}$, b
```

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 algorithms use an iterative method based on mistakes and corrections and finally converge to the optimal result with appropriate hyper-parameters chosen. However, Perceptron algorithm does not produce probability estimates, it is only used to achieve the linear model which predicts the sign of the result. The stochastic gradient descent algorithm, instead, is used to model $P(y = 1|x)$, which serves the goal of logistic regression and can be used for more complicated predictions.

(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:

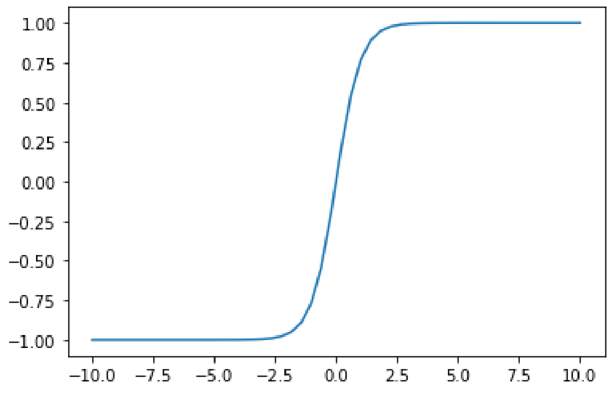


Figure 1: σ_A

σ and σ_A have similar shape but σ_A curves more quickly compared with σ and its value ranges from -1 to 1 instead of 0 to 1. As $z \to \infty$, $\sigma_A = 1$. As $z \to -\infty$, $\sigma_A = -1$.

ii. Prove the following: [5 pts]

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

Solution:

$$\frac{d\sigma_A(z)}{dz} = \frac{(\exp(z) + \exp(-z))^2 - (\exp(z) - \exp(-z))^2}{(\exp(z) + \exp(-z))^2} = 1 - \left(\frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}\right)^2 = 1 - \sigma_A^2(z)$$

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. σ_A ranges from -1 to 1 and probability cannot be negative.

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: $h_A(\boldsymbol{x}_n)$ ranges from 0 to 1 since σ_A ranges from -1 to 1.

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

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

Solution:

$$h_A(\boldsymbol{x}_n) = \frac{1 + \sigma_A(\boldsymbol{w}^T \boldsymbol{x}_n + b)}{2} = \frac{1 + \sigma_A(w_1 x_1^{(n)} + ... + w_m x_m^{(n)} + b)}{2}$$

$$\frac{\partial h_A(\boldsymbol{x}_n)}{\partial w_j} = \frac{1}{2} \frac{\partial \sigma_A(\boldsymbol{w}^T \boldsymbol{x}_n + b)}{\partial(\boldsymbol{w}^T \boldsymbol{x}_n + b)} \frac{\partial(\boldsymbol{w}^T \boldsymbol{x}_n + b)}{\partial w_j} = \frac{1}{2}(1 - (\sigma_A(\boldsymbol{w}^T \boldsymbol{x}_n + b))^2) x_j^n = 2h_A(\boldsymbol{x}_n)(1 - h_A(\boldsymbol{x}_n)) x_j^n$$

$$\frac{\partial J_A}{\partial w_j} = \frac{\partial J_A}{\partial h_A(\boldsymbol{x}_n)} \frac{\partial h_A(\boldsymbol{x}_n)}{\partial w_j} = -\sum_{n=1}^{N} [\frac{y_n}{h_A(\boldsymbol{x}_n)} - \frac{(1 - y_n)}{(1 - h_A(\boldsymbol{x}_n))}] 2h_A(\boldsymbol{x}_n)(1 - h_A(\boldsymbol{x}_n)) x_j^n$$

$$= -\sum_{n=1}^{N} [2y_n(1 - h_A(\boldsymbol{x}_n)) x_j^n - 2(1 - y_n) h_A(\boldsymbol{x}_n) x_j^n] = \boxed{2\sum_{n=1}^{N} [h_A(\boldsymbol{x}_n) x_j^n - y_n x_j^n]}$$

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

$$\frac{\partial J_A}{\partial b} = \frac{\partial J_A}{\partial h_A(\boldsymbol{x}_n)} \frac{\partial h_A(\boldsymbol{x}_n)}{\partial b} = -\sum_{n=1}^{N} [\frac{y_n}{h_A(\boldsymbol{x}_n)} - \frac{(1 - y_n)}{(1 - h_A(\boldsymbol{x}_n))}] 2h_A(\boldsymbol{x}_n)(1 - h_A(\boldsymbol{x}_n))$$

$$= -\sum_{n=1}^{N} [2y_n(1 - h_A(\boldsymbol{x}_n)) - 2(1 - y_n) h_A(\boldsymbol{x}_n)] = \boxed{2\sum_{n=1}^{N} [h_A(\boldsymbol{x}_n) - y_n]}$$

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

Solution: 1. Initialize $\boldsymbol{w}$. Initialize the learning rate r

```
2. For t = 0, 1, 2, .... (until error below some threshold)
      For each training example $(\boldsymbol{x}^i, y^i)$:
         $b^{t+1} = b^t - 2r(h_A(\boldsymbol{x}^i) - y^i)$
         $b^t = b^{t+1}$
         For each element of the weight vector $(w_j)$:
            $w_j^{t+1} = w_j^t - 2r(h_A(\boldsymbol{x}^i)x_j^i - y^i x_j^i)$
            $w_j^t = w_j^{t+1}$
return $\boldsymbol{w}$, b
```

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:

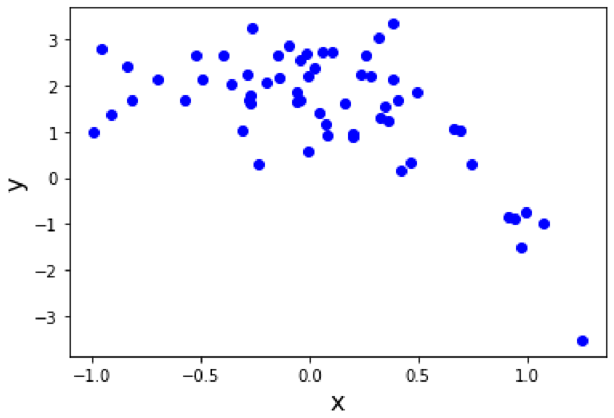


Figure 2: Training Data

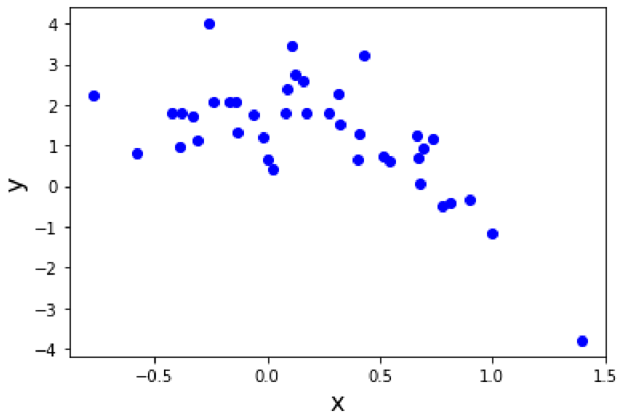


Figure 3: Test Data

The training data and test data are in similar shape, but their trend seems to be more like a polynomial regression instead of linear regression, so linear regression may not be very effective in predicting the data.

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

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:

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

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

(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:

```
first_col = np.ones(shape=(n,1))
Phi = np.append(first_col, X, axis=1)
```

(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 = X.dot(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 $J(\boldsymbol{w})$.

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

In gradient descent, each iteration performs the update

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

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:

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

- **(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:

  ```
  y_pred = X.dot(self.coef_)  # change this line
  gradient = 2 * np.sum((y_pred - y).reshape(n,1)*X, axis=0)
  self.coef_ -= (eta * gradient)
  ```

- **(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:

coefficients	number of iterations	final value	learning rate	time
[1.05869269, -0.35201436]	10000	93.76821090163459	0.000001	0.6921277046203613
[1.58572635, -1.41669696]	10000	60.48308903597063	0.00001	0.5823826789855957
[1.59122863, -1.48402808]	505	60.41076204171135	0.001	0.02992701530456543
[-1.30489153e+111, -1.06471386e+110]	10000	1.0334852857059262e+224	0.0168	0.41951942443847656

  Coefficients for the second and third choice of η are quite close, so does the final values. According to the time taken, the converging speed increases as η increases except for $\eta = 0.0168$. For $\eta = 0.0168$, both the coefficients and final value are extremely larger than all the others, this is probably because the value of η is too large so that the algorithm cannot converge efficiently, the number of iterations already reach the maximum, but the final value is still far away from the expected value.

(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:

```
self.coef_ = np.linalg.pinv(X.T.dot(X)).dot(X.T).dot(y)
return self
```

- **(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: time required for closed form solution: 0.0014193058013916016
  Closed form solution coefficients: [1.59122864 -1.48402822]
  The closed-form solution coefficients is almost the same as the coefficients for learning rate 0.001 in GD. This is expected because the third GD finished in just 505 iterations, its coefficient should be really close to the real one and the time required for the closed-form solution is just 0.0014193058013916016s, which is much faster than all the GD in the table.

(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: number of iterations: 314
coefficients for part f changing learning rate: [1.59122863 -1.48402817]
Coefficients are the same solution yielded by the closed-form optimization and number of iterations is less than all the instances in part(d) with fixed learning rate.

```
if eta_input is None:
    eta = 1/float(2+t)  # number of iterations k=t+1
else:
    eta = eta_input
```

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_
for i in range(2,m+1):
    a = np.array(X**i)
    Phi = np.append(Phi,a,axis=1)
if m==0:
    Phi=first_col
```

(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]

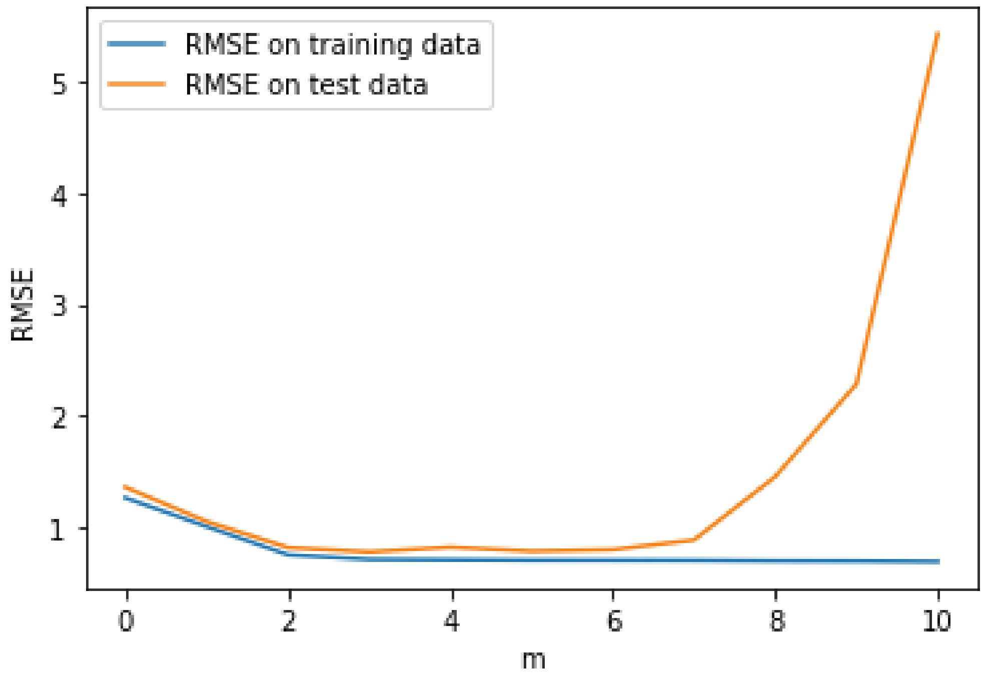
Solution: RMSE normalizes the errors based on the sample size, so it will give a more balanced estimate for the errors of the model regardless of the size of the dataset. Since $J(\boldsymbol{w})$ is influenced by the sample size, RMSE tends to be a better metric.

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

(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:

[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, $k = m+1$.



Degree polynomial 5 best fits the data because RMSE on test data is approximately the lowest when m = 5 by observing the plot and RMSE on training data is also very small when m = 5. When m is greater than or equal to 6, we saw signs of overfitting because RMSE for training data keeps decreasing while RMSE for test data keeps increasing dramatically for $m \geq 6$. The large gap at the end of the plot indicates overfitting. Since RMSE is relatively small(less than one) on both training data and test data at the beginning of the plot, there is no obvious evidence for underfitting.