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 $w_1 X_1 + w_2 X_2 + b$ 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) Two possible sets of values that can implement OR are $(w_1, w_2, b) = (1, 1, 0)$ and $(2, 2, -1)$ (in which the activation function would be a step function f(x) = 1 if x > 0 and 0 otherwise).
(ii) There is no linear model for XOR. If we are to plots all possible input points (0, 0), (0, 1), (1, 0), and (1, 1), we cannot separate (0, 1) and (1, 0) from the other two using only a single line/linear function (if we plot this in 3D coordinates, there is no single plane separating (0, 1, 1) and (1, 0, 1) from (0, 0, 0) and (1, 1, 0)).

(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 distinct possible Boolean functions possible with 2 Boolean variables (2^4 possible output combinations). Out of these 16, only 0, $NOR(X_1, X_2)$, $AND(X_1, \bar{X}_2)$, $AND(\bar{X}_1, X_2)$, $AND(X_1, X_2)$, $NOT(X_1)$, $NOT(X_2)$, X_1, X_2, $OR(X_1, X_2)$, $NAND(X_1, \bar{X}_2)$, $NAND(\bar{X}_1, X_2)$, and $NAND(X_1, X_2)$ can be represented with a linear model.

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:

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

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:
$\frac{\partial J}{\partial w_j} = \sum_{n=1}^{N}((-y_n + \sigma(\boldsymbol{w}^T \boldsymbol{x}_n + b))x_n)$
$\frac{\partial J}{\partial b} = \sum_{n=1}^{N}(y_n - \sigma(\boldsymbol{w}^T \boldsymbol{x}_n + b))$

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

Solution:

```
Initialize W to 0 $\epsilon R^N$ and b to 0
For each T = {0, 1, 2, ...}:
    For each (x, y) in dataset D:
        Calculate $\frac{\partial J}{\partial w_j}$ and $\frac{\partial J}{\partial b}$ for the given x and y
        W$^{T+1} = W^T - \eta_1 \frac{\partial J}{\partial w_j}$
        b$^{T+1} = b^T - \eta_2 \frac{\partial J}{\partial b}$
```

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: The biggest key difference between the two is that for the perceptron algorithm, because the output is a hard 0 or 1 (in contrast to the rangeed output from 0 to 1 for logistic regression), the update for preceptron only updates the current values of W and b if the output of f(Wx+b) doesn't agree with the training output y for a specific training case.

(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]

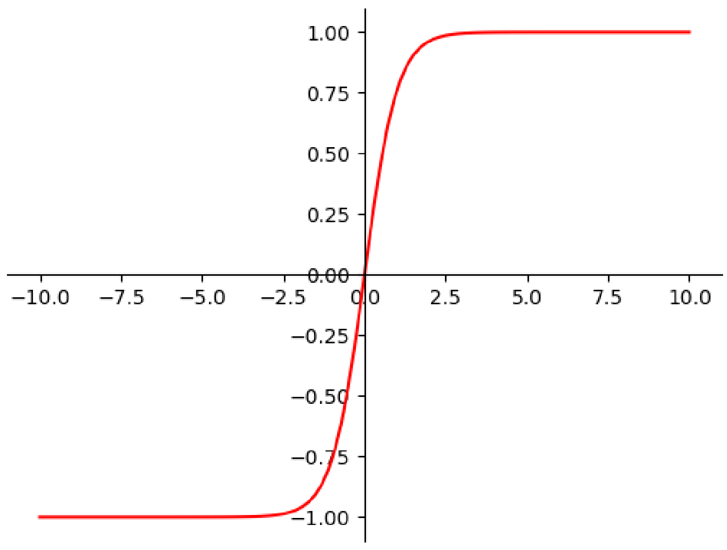
Solution:
CODE:

```
import matplotlib.pyplot as plt
import numpy as np
import math

def func(x):
    y = np.array([])
    for val in x:
        posExp = math.exp(val)
        negExp = math.exp(val * -1.0)
        expVal = (posExp - negExp) / (posExp + negExp)
        y = np.append(y, expVal)
    return y

x = np.linspace(-10, 10, 100)
y = func(x)

fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.spines['left'].set_position('center')
ax.spines['bottom'].set_position('zero')
plt.plot(x,y, 'r')
plt.show()
```



The main similarities between the two functions is that they both have the same general shape. However, the new function σ_A hase a range of (-1, 1) as opposed to (0, 1) for σ. The range of the former is twice that of the latter, and is also an odd function.

ii. Prove the following: [5 pts]

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

Solution:

$$\begin{aligned} \frac{d\sigma_A(z)}{dz} &= \frac{d}{dz}(\frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}) \\ &= \frac{(\exp(z)+\exp(-z))(\exp(z)+\exp(-z))-(\exp(z)-\exp(-z))(\exp(z)-\exp(-z))}{(\exp(z)+\exp(-z))^2} \\ &= \frac{4}{(\exp(z)+\exp(-z))^2} \\ &= \frac{4\exp(z)\exp(-z)}{(\exp(z)+\exp(-z))^2} \end{aligned}$$

$$
\begin{aligned}
&= \left(\frac{2\exp(z)}{\exp(z)+\exp(-z)}\right)\left(\frac{2\exp(-z)}{\exp(z)+\exp(-z)}\right)\\
&= (1+\sigma_A(z))(1-\sigma_A(z))\\
&= 1-\sigma_A^2(z)
\end{aligned}
$$

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; the most obvious reason is that we cannot have a negative probability (which is the case for $\sigma_A(z)$ if z <= 0).

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

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

Solution:
$\frac{\partial J_A}{\partial w_j} = \sum_{n=1}^N 2(y_n - h_A(x_n))x_n$
$\frac{\partial J_A}{\partial b} = \sum_{n=1}^N 2(y_n - h_A(x_n))$

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

Solution:

```
Initialize W to 0 $\epsilon R^N$ and b to 0
For each T = {0, 1, 2, ...}:
    For each (x, y) in dataset D:
        Calculate $\frac{\partial J_A}{\partial w_j}$ and $\frac{\partial J_A}{\partial b}$ for the given x and y
        W$^{T+1} = W^T - \eta_1 \frac{\partial J}{\partial w_j}$
        b$^{T+1} = b^T - \eta_2 \frac{\partial J}{\partial b}$
```

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 $y_n \in \mathbb{R}, n \in \{1, \dots, N\}$, 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]

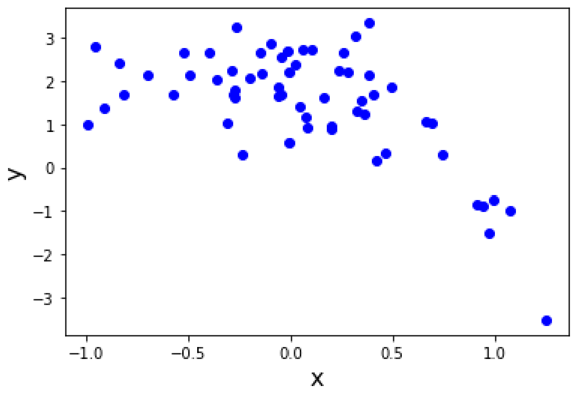
[1] Try out `SciPy`'s tutorial (http://wiki.scipy.org/Tentative_NumPy_Tutorial), or use your favorite search engine to find an alternative. Those familiar with Matlab may find the "Numpy for Matlab Users" documentation (http://wiki.scipy.org/NumPy_for_Matlab_Users) more helpful.

[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 data seems to be close to horizontally linear for x = 0 to 0.5, but seems to curve downwards for x ¿ 5. So, due to the curve, linear regression might not be able to fit the model perfectly (since the curve seems to indicate a polynomial relation between x and 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 $\boldsymbol{x}_n = \left(1, x_{n,1}, \dots, x_{n,D}\right)^T$.

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: (see code for implementation)

(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: (see code for implementation)

(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: (see code for implementation)

- **(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: (see code for implementation)

- **(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:

η	Coefficients	Iterations	Final Error
0.000001	[1.05869269 -0.35201436]	10000	93.76821090163463
0.00001	[1.58572635 -1.41669696]	10000	60.48308903597062
0.001	[1.59122862 -1.48402808]	503	60.4107620417114
0.0168	[-1.30489153e+111 -1.06471386e+110]	10000	1.0334852857059493e+224

  For 10^{-6} and 10^{-5}, the step sizes were too small, although there were still converging, so the limit of 10000 iterations were hit. For 10^{-3}, the convergence stops at about 503 iterations. For 0.0168 and larger values, the step size is too large and the function will never converge (and even end up with increasing errors the more iterations pass).

(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: (see code for implementation)

- **(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:
  Coefficients: [1.59122864 -1.48402822]
  Cost: 60.41076204171104
  The algorithm runs much quicker than GD (and doesn't depend on multiple iterations required for GD), as well have having a smaller error cost.

(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: 357 iterations

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: (see code for implementation)

(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 $J(\boldsymbol{w})$?

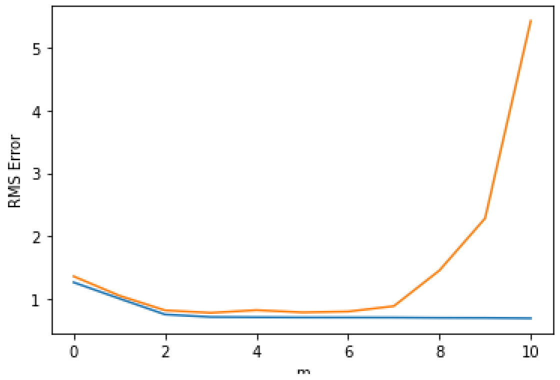
Implement `PolynomialRegression.rms_error(...)`. [4 pts]

Solution:
RMSE might be preferred over J(w) since RMSE scales likely less per sample as it operates on a square root scale (meaning more samples wouldn't increase the error as much). In addition, it also attempts to normalize the error with the number of samples N (as a greater number of cases in J(w) would mean an increased error, which might not be the case due to increasing variance and increasing number of offset errors to factor in).

(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]

[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, $k = m+1$.

Solution:



Based on the resulting RMSE for the testing data, m = 3 would likely be the best fit. There is definitely evidence of overfitting, as the training error decreases as m increases, but the testing error increases after a certain threshold.