University of California, Los Angeles
Cyrus Asasi
506014946

ECE M146 **Prof. Suhas Diggavi**

Homework 6

Due Wednesday, April 24th, 2024 11:59pm **via Gradescope**

1. Suppose we have a training set with 8 samples, each sample has feature vector in $\mathbb{R}^2$:

#	1	2	3	4	5	6	7	8
X	[4,0]	[1,1]	[0,1]	[-2,-2]	[-2,1]	[1,0]	[5,2]	[3,0]
y	1	-1	-1	1	-1	1	-1	-1

We are going to implement the perceptron algorithm to train a linear classifier with 2 dimensional weight vector $\mathbf{w} \in \mathbb{R}^2$ (no bias term). We start with initial weight vector as the first sample in our dataset, i.e. $\mathbf{w}_1 = \mathbf{x}_1$. Note that: when $\mathbf{w}^\top x = 0$, the algorithm predicts $+1$.

To simplify the calculation, you only need to test and possibly update each sample once in the given sequence. You can either implement the algorithm by hand or programming.

(a) Is the data linearly separable? Will our algorithm converge if we run it several times over the same sequence? Explain.

By observing figure 1, we can clearly see that the dataset is not linearly separable. Since the dataset is not linearly separable, the perceptron learning algorithm will not converge. This can be illustrated by figure 2, which was generated by running the perceptron algorithm 100 times on the data set and assigning a newly colored line corresponding to each updated vector $\mathbf{w}$.

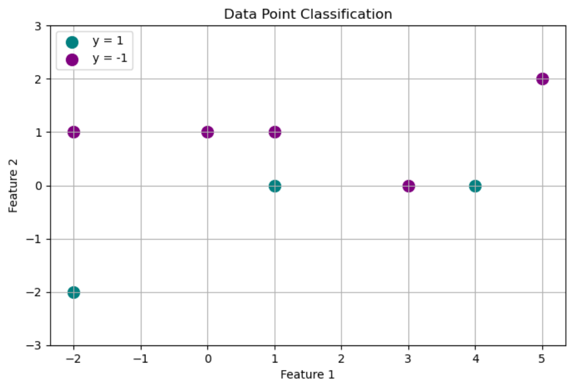

Figure 1: Plot of dataset.

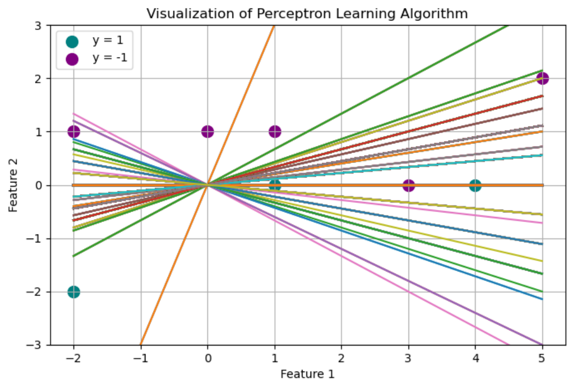

Figure 2: Plot of perception learning algorithm run 100 times.

(b) Regardless of whether the dataset is linearly separable or not, calculate the updates of the weight vector on this sequence for one round over the entire dataset. Follow the order of the index for the samples and show your calculations.

$$
\begin{aligned}
w_0 &= [4,0] \quad \text{initial}\\
w_1 &= [4,0] \quad \text{no update}\\
w_2 &= [4,0] + -1 * [1,1] = [3,-1] \quad \text{update}\\
w_3 &= [3,-1] \quad \text{no update}\\
w_4 &= [3,-1] + 1 * [-2,-2] = [1,-3] \quad \text{update}\\
w_5 &= [1,-3] \quad \text{no update}\\
w_6 &= [1,-3] \quad \text{no update}\\
w_7 &= [1,-3] \quad \text{no update}\\
w_8 &= [1,-3] + -1 * [3,0] = [-2,-3] \quad \text{update}
\end{aligned}
$$

(c) Provide closed-form functions for the perceptron, Voted perceptron, and Average perceptron, using the weight vector(s) derived in part (b).

i. Perceptron: The perceptron algorithm updates the weight vector when a mistake is made. We can use the final weight vector to construct a closed-form function for future predictions.

$$f(\mathbf{x}) = sign(\mathbf{w}^T\mathbf{x}) = sign([-2,-3]\cdot\mathbf{x})$$

ii. Voted Perceptron: The voted perceptron maintains a list of weight vectors along with the number of times each one 'voted' for a correct classification before an update was needed.

$$
\begin{aligned}
f(\mathbf{x}) &= sign\left[\sum_{i=1} s_i \cdot sign(\mathbf{w}_i^T\mathbf{x})\right]\\
&= sign\left[2\cdot sign([4,0]\cdot\mathbf{x}) + 2\cdot sign([3,-1]\cdot\mathbf{x}) + 4\cdot sign([1,-3]\cdot\mathbf{x}) + 1\cdot sign([-2,-3])\cdot\mathbf{x}\right]
\end{aligned}
$$

iii. Average Perceptron: The average perceptron takes the average of all the weight vectors across updates. This weighted average vector is:

$$
\begin{aligned}
f(\mathbf{x}) &= sign\left[\sum_{i=1} s_i \cdot \mathbf{w}_i^T\mathbf{x}\right]\\
&= sign\left[2\cdot[4,0]\cdot\mathbf{x} + 2\cdot[3,-1]\cdot\mathbf{x} + 4\cdot[1,-3]\cdot\mathbf{x} + 1\cdot[-2,-3]\cdot\mathbf{x}\right]
\end{aligned}
$$

(d) Using the functions derived in part (c), compare the errors between the perceptron, the Voted perceptron predictor, and the Average perceptron predictor across the entire dataset. For each point in the dataset, find the label assigned by each classifier and report the error over the dataset.

y_i is perceptron, y_{ii} is voted, y_{iii} is averaged

#	1	2	3	4	5	6	7	8
X	[4,0]	[1,1]	[0,1]	[-2,-2]	[-2,1]	[1,0]	[5,2]	[3,0]
y	1	-1	-1	1	-1	1	-1	-1
y_i	-1	-1	-1	1	1	-1	-1	-1
y_{ii}	1	-1	-1	1	-1	1	-1	1
y_{iii}	1	-1	-1	1	-1	1	1	1

```
Perceptron Error Rate: 0.375
Voted Perceptron Error Rate: 0.125
Average Perceptron Error Rate: 0.25
```

2. In class, we have seen the logistic regression when labels are $\{0,1\}$. In this question, you will derive the logistic regression when labels are instead $\{-1,1\}$.

In class, we considered the dataset $\mathcal{D} = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ with n samples where $\mathbf{x}_i \in \mathbb{R}^d$ and labels $y_i \in \{0,1\}$ for all $i \in [n]$. The prediction function $h_{\mathbf{w}}(\mathbf{x})$ studied in class is given by

$$h_{\mathbf{w}}(\mathbf{x}) = \frac{1}{1+e^{-\mathbf{w}^\top \mathbf{x}}} = \frac{e^{\mathbf{w}^\top \mathbf{x}}}{1+e^{\mathbf{w}^\top \mathbf{x}}} \tag{1}$$

Moreover, the objective studied in class to minimize for logistic regression was:

$$\text{P2.1:} \qquad \min_{\mathbf{w}} -\sum_{i=1}^{n} \left[y_i \log\left(h_{\mathbf{w}}(\mathbf{x}_i)\right) + (1-y_i)\log\left(1-h_{\mathbf{w}}(\mathbf{x}_i)\right)\right] \tag{2}$$

We want to modify this objective function so that $\tanh_{\mathbf{w}}(\mathbf{x}) = \frac{e^{\mathbf{w}^\top \mathbf{x}} - e^{-\mathbf{w}^\top \mathbf{x}}}{e^{\mathbf{w}^\top \mathbf{x}} + e^{-\mathbf{w}^\top \mathbf{x}}} = \frac{e^{2\mathbf{w}^\top \mathbf{x}}-1}{e^{2\mathbf{w}^\top \mathbf{x}}+1}$ is the activation function and $\tilde{y}_i \in \{-1,1\}$ the labels.

(a) Show that

$$\tanh_{\mathbf{w}}(\mathbf{x}) = 2h_{\mathbf{w}'}(\mathbf{x}) - 1, \mathbf{w}' = 2\mathbf{w}$$

$$h_{\mathbf{w}'}(\mathbf{x}) = h_{2\mathbf{w}}(x) = \frac{e^{2\mathbf{w}^T\mathbf{x}}}{1+e^{2\mathbf{w}^T\mathbf{x}}}$$

$$2h_{\mathbf{w}'}(\mathbf{x}) - 1 = \frac{2e^{2\mathbf{w}^T\mathbf{x}}}{1+e^{2\mathbf{w}^T\mathbf{x}}} - \frac{1+e^{2\mathbf{w}^T\mathbf{x}}}{1+e^{2\mathbf{w}^T\mathbf{x}}} = \frac{e^{2\mathbf{w}^\top\mathbf{x}}-1}{e^{2\mathbf{w}^\top\mathbf{x}}+1}$$

Therefore, $\tanh_{\mathbf{w}}(\mathbf{x}) = 2h_{\mathbf{w}'}(\mathbf{x}) - 1$

(b) What are the asymptotic values of the function $\tanh_{\mathbf{w}}(\mathbf{x})$ as $\mathbf{w}^\top\mathbf{x} \to \infty$ and $\mathbf{w}^\top\mathbf{x} \to -\infty$? Roughly draw the graph of this function with respect to $\mathbf{w}^\top\mathbf{x}$. What is the decision criterion you can choose for predicting labels as -1 or 1?

The asymptotic values of the $tanh_{\mathbf{w}}(\mathbf{x})$ as $\mathbf{w}^\top\mathbf{x} \to \infty$ and $\mathbf{w}^\top\mathbf{x} \to -\infty$ are 1 and -1 respectively.

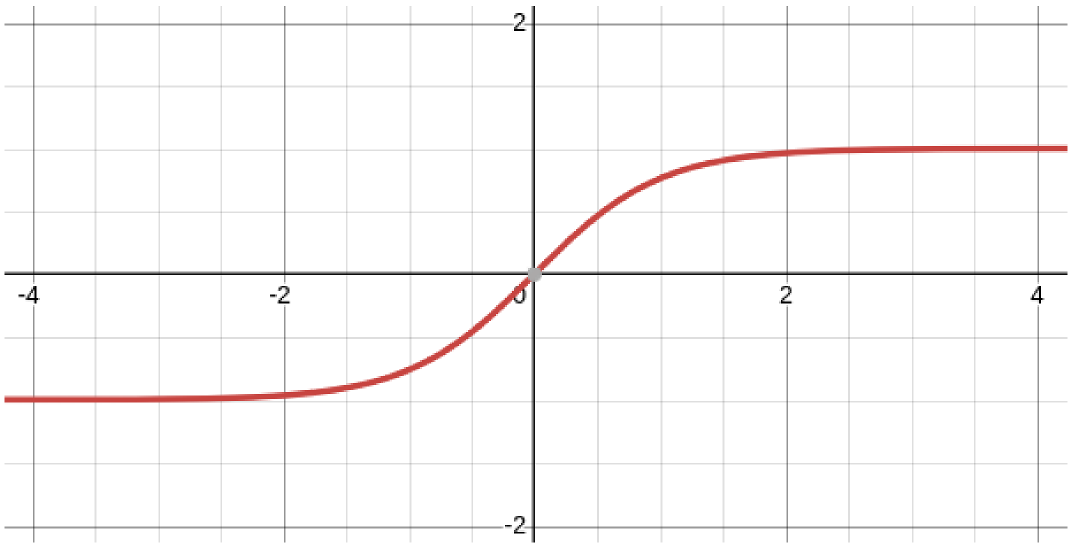

Figure 3: Graph of $tanh_{\mathbf{w}}(\mathbf{x})$

A good decision criteria for predicting lables as -1 or 1 can be defined as follows.

$$\text{label classifier} = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

(c) Using your answer in part (b), argue that we cannot directly replace $h_{\mathbf{w}}(\mathbf{x}_i)$ with $\tanh_{\mathbf{w}}(\mathbf{x})$ in the optimization problem P2.1.

We cannot use $tanh_{\mathbf{w}}(\mathbf{x})$ to replace $h_{\mathbf{w}}(\mathbf{x}_i)$ because logarithms of negative numbers are not defined. While $h_{\mathbf{w}}(\mathbf{x}_i)$ remains positive, $tanh_{\mathbf{w}}(\mathbf{x})$ drops below zero when $x < 0$. This makes it impossible to use $tanh_{\mathbf{w}}(\mathbf{x})$ in the optimization problem P2.1.

(d) When labels are $\tilde{y}_i \in \{-1, 1\}$ show, using your answer in part (a), that the optimization problem in P2.1 is equivalent to:

$$\text{P2.2:} \quad \min_{\mathbf{w}} -\sum_{i=1}^{n}\left[\frac{1+\tilde{y}_i}{2}\log\left(\frac{1+\tanh_{\mathbf{w}}(\mathbf{x}_i)}{2}\right)+\frac{1-\tilde{y}_i}{2}\log\left(\frac{1-\tanh_{\mathbf{w}}(\mathbf{x}_i)}{2}\right)\right] \tag{3}$$

$\frac{1+\tilde{y}_i}{2}$ and $\frac{1-\tilde{y}_i}{2}$ essentially modify P2.2 so that it has the same behavior as the y_i and $(1-y_i)$ in P2.1, except that it works when the values of y are 1 and -1, rather than 1 and 0.

To show that P2.2 for $\tilde{y}_i \in \{-1, 1\}$ is the same as P2.1, we also need to show that the following two identities hold. However, both of these have already been proven in part a). By substituting $\tanh_{\mathbf{w}}(\mathbf{x}) = 2h_{\mathbf{w}'}(\mathbf{x}) - 1$, we can clearly see that both identities hold.

$$\log\left[\frac{1+\tanh_{\mathbf{w}}(\mathbf{x}_i)}{2}\right] = \log\left(h_{\mathbf{w}}(\mathbf{x}_i)\right)$$
$$\log\left[\frac{1-\tanh_{\mathbf{w}}(\mathbf{x}_i)}{2}\right] = \log\left(1-h_{\mathbf{w}}(\mathbf{x}_i)\right)$$

(e) Compute the gradient of the loss function in P2.2 (3) for a single sample $\mathbf{x}_i$. Consider the two cases $\tilde{y}_i = 1$ and $\tilde{y}_i = -1$ separately.

For $\tilde{y}_i = 1$:

$$L(\mathbf{w}) = -\left[\log\left(\frac{1+\tanh(\mathbf{w}^\top\mathbf{x}_i)}{2}\right)\right]$$

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{w}} &= -\left(\frac{2}{1+\tanh(\mathbf{w}^\top\mathbf{x}_i)}\right)\left(\frac{1-\tanh^2(\mathbf{w}^\top\mathbf{x}_i)}{2}\right)\frac{\partial}{\partial \mathbf{w}}(\mathbf{w}^\top\mathbf{x}_i)\\ &= -\left(\frac{1-\tanh^2(\mathbf{w}^\top\mathbf{x}_i)}{1+\tanh(\mathbf{w}^\top\mathbf{x}_i)}\right)\mathbf{x}_i\end{aligned}$$

For $\tilde{y}_i = -1$:

$$L(\mathbf{w}) = -\left[\log\left(\frac{1-\tanh(\mathbf{w}^\top\mathbf{x}_i)}{2}\right)\right]$$

$$\begin{aligned}\frac{\partial L}{\partial \mathbf{w}} &= -\left(\frac{2}{1-\tanh(\mathbf{w}^\top\mathbf{x}_i)}\right)\left(\frac{1-\tanh^2(\mathbf{w}^\top\mathbf{x}_i)}{2}\right)\frac{\partial}{\partial \mathbf{w}}(\mathbf{w}^\top\mathbf{x}_i)\\ &= -\left(\frac{1-\tanh^2(\mathbf{w}^\top\mathbf{x}_i)}{1-\tanh(\mathbf{w}^\top\mathbf{x}_i)}\right)\mathbf{x}_i\end{aligned}$$

3. In each of these problems, give a clear explanation for your choice.

(a) **Perceptrons**:

i. When the perceptron algorithm encounters an incorrectly classified sample $(\mathbf{x}_i, y_i)$ with weights $\mathbf{w}$, it updates the weights and obtains $\mathbf{w}'$ after the update. $\mathbf{w}'$ correctly classifies $(\mathbf{x}_i, y_i)$.

FALSE. The perceptron algorithm does not guarantee that $\mathbf{w}'$ will be correct after the update. It only guarantees that $\mathbf{w}'$ will be closer to correctly classifying a sample.

ii. If the data is linearly seperable, then the Rosenblatt perceptron algorithm converges to a solution that makes no errors on the training data.

TRUE. The perceptron algorithm will eventually converge to a solution that correctly classifies all points in a linearly seperable dataset because it iteratively adjusts the weights and bias to correctly classify all data points.

iii. For a perceptron classifier with weights $\mathbf{w}$, the prediction made on a feature $\mathbf{x}$ is $sign\left(\mathbf{w}^\top \mathbf{x}\right)$. If we multiply the weights obtained from perceptron by a negative scalar, this flips all predictions made by the perceptron algorithm.

FALSE. Simply multiplying the weights by a negative scalar will not flip all predictions made by the perceptron algorithm because if $\mathbf{w}^T\mathbf{x}$ happens to be 0, multiplying it by a negative scalar will not change the prediction.

(b) **Logistic Regression**:

i. The prediction of a logistic regression classifier is given by $y = \sigma\left(\mathbf{w}^\top \mathbf{x}\right) \in [0, 1]$. Moreover, the decision boundary of this classifier is $\mathbf{w}^\top \mathbf{x} \geq 0$.

TRUE. The decision boundary is when $\sigma \mathbf{w}^T\mathbf{x} = 0.5$ which corresponsds to when $\mathbf{w}^T\mathbf{x} = 0$. The sigmoid function works as defined by the problem, and is defined on [0, 1].

ii. The stochastic gradient descent optimization is more computationally efficient than batch gradient descent per iteration.

TRUE. This is definitely the case because stochastic gradient descent chooses a single random point rather than considering all points when calculating loss and derivative.

iii. Let $h_\mathbf{w}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x}, (\mathbf{x}, y)$ a sample, and $\sigma(a) = \frac{1}{1+e^{-a}}$. When $y = 0$, it is true that

$$-y \log\left(\sigma\left(h_\mathbf{w}(\mathbf{x})\right)\right) - (1-y)\log\left(1 - \sigma\left(h_\mathbf{w}(\mathbf{x})\right)\right) = \log\left(1 + e^{h_\mathbf{w}(\mathbf{x})}\right) - 1_{y=1} h_\mathbf{w}(\mathbf{x})$$

TRUE. When $y = 0$

$$\begin{aligned}
-y \log\left(\sigma\left(h_\mathbf{w}(\mathbf{x})\right)\right) - (1-y)\log\left(1 - \sigma\left(h_\mathbf{w}(\mathbf{x})\right)\right) &= -\log\left(1 - \sigma\left(h_\mathbf{w}(\mathbf{x})\right)\right) \\
&= -\log\left(1 - \frac{1}{1+e^{-h_\mathbf{w}(\mathbf{x})}}\right) \\
&= -\log\left(\frac{e^{-h_\mathbf{w}(\mathbf{x})}}{1+e^{-h_\mathbf{w}(\mathbf{x})}}\right) \\
&= -\left[\log(e^{-h_\mathbf{w}(\mathbf{x})}) - \log(1+e^{-h_\mathbf{w}(\mathbf{x})})\right] \\
&= h_\mathbf{w}(\mathbf{x}) + \log(1+e^{-h_\mathbf{w}(\mathbf{x})}) \\
&= \log(1+e^{h_\mathbf{w}(\mathbf{x})}) \\
&= \log(1+e^{h_\mathbf{w}(\mathbf{x})}) - 1_{y=1} h_\mathbf{w}(\mathbf{x})
\end{aligned}$$

$1_{y=1} = 0$ when $y = 0$

iv. Let $\sigma(a) = \frac{1}{1+e^{-a}}$, then the derivative is $\frac{d\sigma(a)}{da} = \frac{-1+e^{-a}}{(1+e^{-a})^2}$.

FALSE. The derivative of the sigmoid function $\sigma(a)$ is

$$\sigma(a) \cdot (1 - \sigma(a)) \implies \frac{1}{1+e^{-a}} \cdot \left(1 - \frac{1}{1+e^{-a}}\right) = \frac{e^{-a}}{(1+e^{-a})^2} \neq \frac{-1+e^{-a}}{(1+e^{-a})^2}$$

(c) **Linear Regression**:

i. A closed form solution for linear regression is only possible if $\boldsymbol{X}^\top \boldsymbol{X}$ is positive definite. Note that a matrix $\boldsymbol{A}$ is positive definite if $\mathbf{z}^\top \boldsymbol{A}\mathbf{z} > 0$ for all vectors $\mathbf{z} \neq \mathbf{0}$

FALSE. A closed form solution for linear regression is possible if $\mathbf{X}^T\mathbf{X}$ is positive semi-definite and invertible.

ii. The loss function of ℓ_2 regularized linear regression is $\|\boldsymbol{X}\mathbf{w} - \mathbf{y}\|_2^2 + \lambda\|\mathbf{w}\|_2^2$. The gradient of the loss function with respect to $\mathbf{w}$ is given by $2\boldsymbol{X}^\top(\boldsymbol{X}\mathbf{w} - \mathbf{y}) + 2\lambda\mathbf{w}$.

TRUE. The derivative computation below shows the validity of the above statement.

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{w}}\|\boldsymbol{X}\mathbf{w}-\mathbf{y}\|_2^2+\frac{\partial}{\partial \mathbf{w}}\lambda\|\mathbf{w}\|_2^2 &= \frac{\partial}{\partial \mathbf{w}}\left[(\boldsymbol{X}\mathbf{w}-\mathbf{y})^\top(\boldsymbol{X}\mathbf{w}-\mathbf{y})\right]+\frac{\partial}{\partial \mathbf{w}}\lambda(\mathbf{w}^T\mathbf{w})\\
&= \frac{\partial}{\partial \mathbf{w}}\left[(\mathbf{X}^T\mathbf{w}^T-\mathbf{y}^T)(\mathbf{X}\mathbf{w}-\mathbf{y})\right]+2\lambda\mathbf{w}\\
&= \frac{\partial}{\partial \mathbf{w}}\left[\mathbf{w}^\top\boldsymbol{X}^\top\boldsymbol{X}\mathbf{w}-\mathbf{w}^\top\boldsymbol{X}^\top\mathbf{y}-\mathbf{y}^\top\boldsymbol{X}\mathbf{w}+\mathbf{y}^\top\mathbf{y}\right]+2\lambda\mathbf{w}\\
&= 2\mathbf{X}^T\mathbf{X}\mathbf{w}-2\mathbf{X}^T\mathbf{y}+0+2\lambda\mathbf{w}\\
&= 2\mathbf{X}^T(\mathbf{X}\mathbf{w}-\mathbf{y})+2\lambda\mathbf{w}
\end{aligned}$$

4. In this exercise, you will work through a family of binary classifications. Our data consists of inputs $x_n \in \mathbb{R}^{1\times d}$ and labels $y_n \in \{-1, 1\}$ for $n \in \{1, \ldots, N\}$. We will work on a subset of the Fashion-MNIST dataset which focuses on classifying whether the image is for a Dress ($y = 1$) or a Shirt ($y = -1$).Your goal is to learn a classifier based on linear predictor $h_{\mathbf{w}}(x) = \mathbf{w}^T x$. Let

$$\mathbf{X} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \in \mathbb{R}^{N\times d}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} \in \{1, -1\}^N \tag{4}$$

The main file is the Notebook Jupyter notebook.

(a) (**Visualization**): Visualize a sample of the training data. What is the dimensions of X_{train} , and X_{test} .

The dimensions of $X_{\text{train}} = (5000, 784)$ and the dimensions of $X_{\text{test}} = (500, 784)$.

(b) (**Perceptron**): Implement Perceptron Algorithm to classify your training data. Let the maximum number of iterations of the Algorithm num $_{\text{iter}}$ $= N$ (number of training samples). At each iteration, compute the percentage of misclassified points in the training dataset, and save it into a `Loss_hist` array. Plot the history of the loss function (`Loss_hist`). What is the final value of the loss function and the squared ℓ_2 norm value of the weight $\|\mathbf{w}\|_2^2$? Looking at the loss function, can you comment on whether the Perceptron algorithm converges?

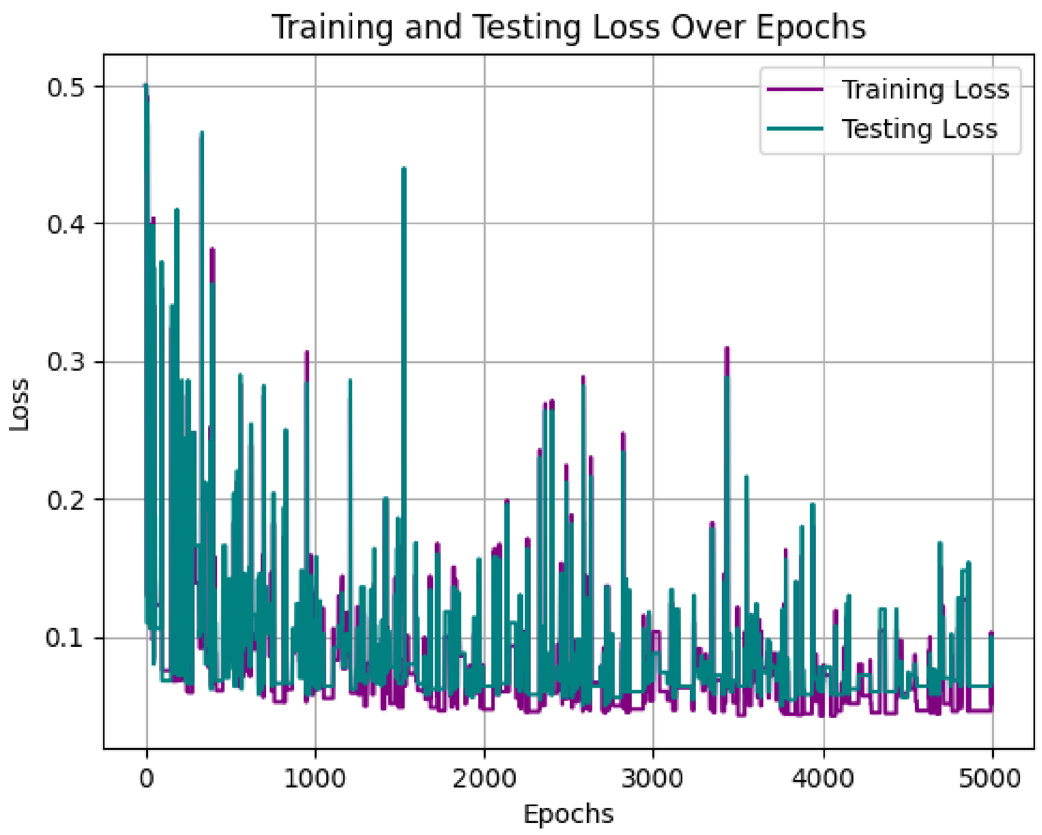

Figure 4: Perceptron Training Loss Over Iterations

The final value of the loss function was 5.16% missclassified, while the final $\ell 2$ norm value of the weight vector was 1404179381. From Figure 4 and the loss function, we can clearly see that the perceptron algorithm does not converge.

(c) (**Perceptron test error**): Compute the percentage of misclassified points in the test data for the trained Perceptron. The percentage of misclassified points in the test data after training the perceptron is 6.80%.

(d) (**Logistic Regression**): In this part, we will implement the logistic regression for binary classification. Recall that logistic regression attempts to minimize the objective function

$$J(\mathbf{w}) = \frac{1}{N}\left(\sum_{n=1}^{N} \log\left(1+e^{h_{\mathbf{w}}(\mathbf{x}_n)}\right) - \sum_{n=1}^{N} \mathbf{1}_{y_n=1} h_{\mathbf{w}}(\mathbf{x}_n)\right) \tag{5}$$

where $\mathbf{x}_n = (1, x_n)$, and $\mathbf{1}_A = 1$ if A is true and 0 otherwise. Moreover, $h_{\mathbf{w}}(\mathbf{x}_n) = \mathbf{w}^T\mathbf{x}_n$. First, we will add an additional feature to each instance and set it to one. This is equivalent to adding an additional first column to $\mathbf{X}$ and setting it to all ones. Modify the `get_features()` in `Logistic.py` file to create a matrix $\mathbf{X}$ for logistic regression model.

(e) Complete predict() in `Logistic.py` file to predict $\mathbf{y}$ from $\mathbf{X}$ and $\mathbf{w}$.

(f) Complete the function `loss_and_grad()` to compute the loss function and the gradient of the loss function with respect to $\mathbf{w}$ for a data set $\mathbf{X}$ and labels $\mathbf{y}$ at given weights $\mathbf{w}$. Test your results by running the code in the main file `Notebook.ipynb`. If you implement everything correctly, you should get the loss function within 0.7 and squared ℓ_2 norm of the gradient around 1.8×10^5.

(g) Complete the function `train_LR()` to train the logistic regression model for given learning rate $\eta = 10^{-6}$, `batch_size` $= 100$, and number of iterations num $_{\text{iters}}$ $= 5000$. Plot the history of the loss function (`Loss_hist`). What is the final value of the loss function and the squared ℓ_2 norm value of the weight $\|\mathbf{w}\|_2^2$?

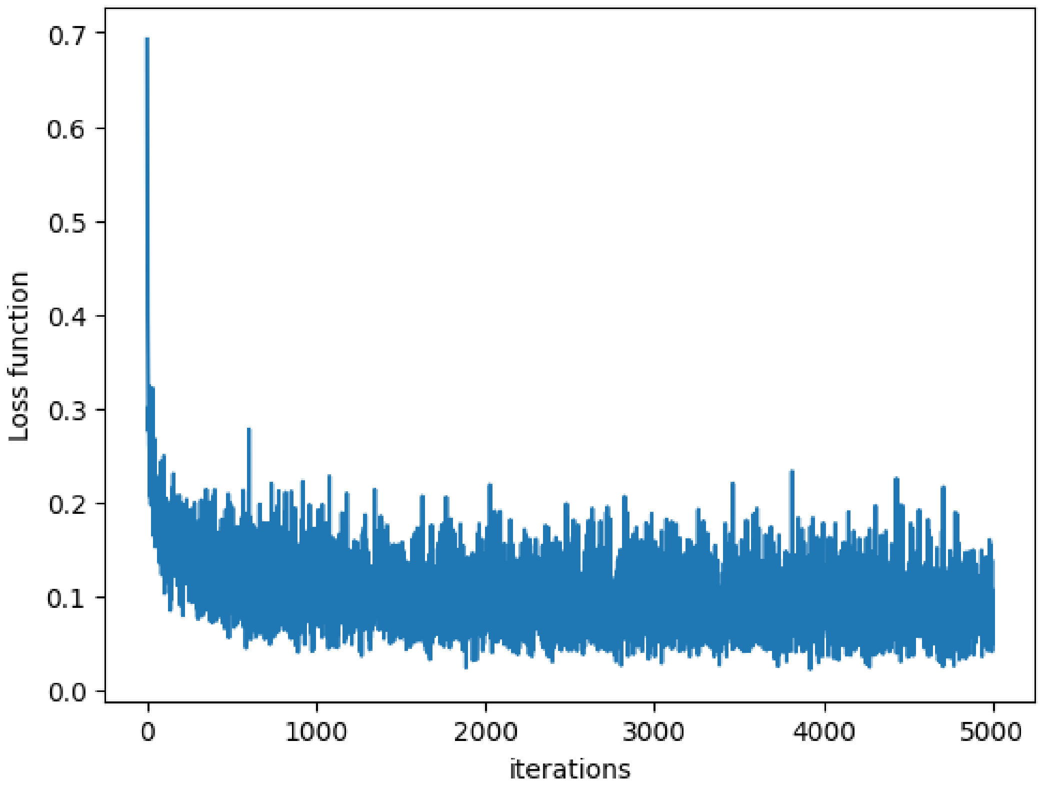

Figure 5: History of the Loss Function

The final value of the loss function is 0.0694
The squared ℓ_2 norm value of the weight is 0.000287

(h) (**Logistic Regression test error**): Compute the percentage of misclassified points in the test data for the trained Logistic Regression.
The percentage of misclassified points in the test data calculated by the predict function is 6.6%

(i) (**Logistic Regression and Batch Size**): Train the Logistic regression model with different batch size $b \in \{1, 50, 100, 200, 300\}$, learning rate $\eta = 10^{-5}$, and number of iterations $num_{iter} = 6000/b$. Train each model 50 or 100 times and average the test error for each value of batch size. Plot the test error as a function of the batch size. Which batch size gives the minimum test error?

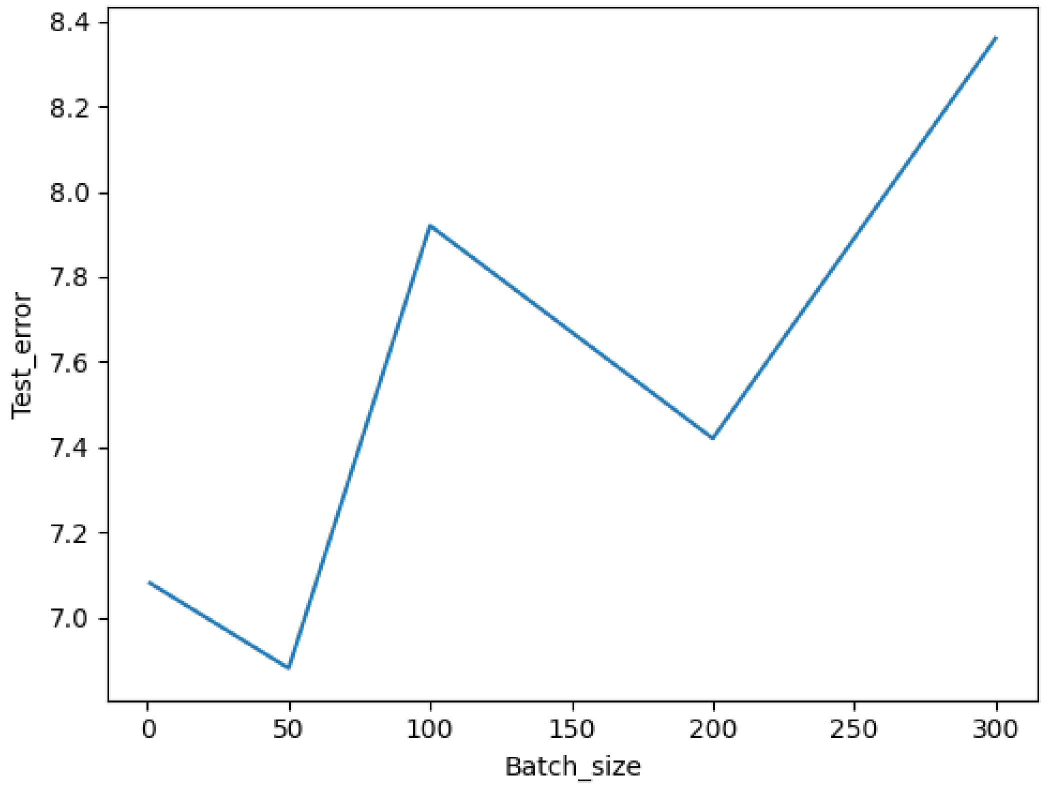

Figure 6: Test Error for Batch Logistic Regression

From the graph of the error, we can clearly see that the batch size of 50 gives the minimum error.

5. 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 $h_{\mathbf{w}}(x)$ that best approximates $f(x)$.

Visualization

As we learned last week, 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).

Linear Regression

Recall that linear regression attempts to minimize the objective function

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

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

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

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

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:

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} = w_0 + w_1 x_1$$

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

(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?

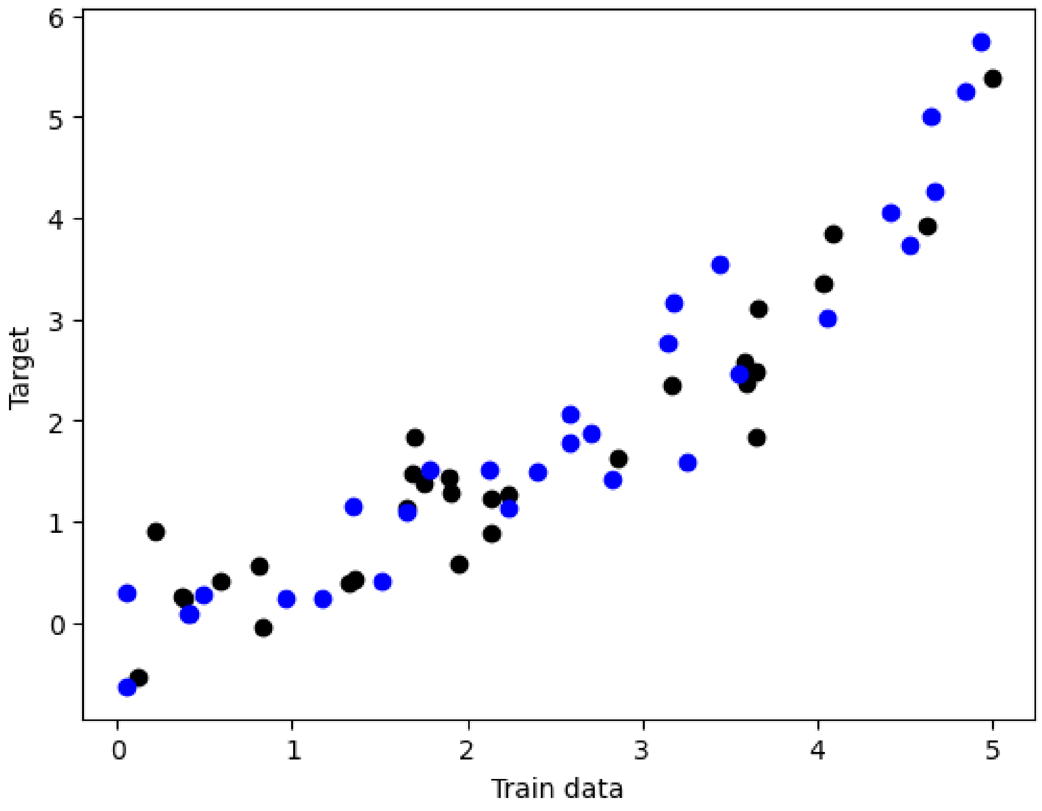

Figure 7: Visualization of Data For Linear Regression

The scatter plot shows a nonlinear, possibly polynomial or exponential trend. Linear regression may not effectively predict this data due to its non-linearity; a nonlinear model would likely be more appropriate.

(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. Modify `get_poly_features()` in `Regression.py` for the case $m = 1$ to create the matrix $\boldsymbol{X}$ for a simple linear model.

See code in appendix

(c) Before tackling the harder problem of training the regression model, complete predict() in `Regression.py` to predict $\mathbf{y}$ from $\boldsymbol{X}$ and $\mathbf{w}$.

See code in appendix

(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 $J(\mathbf{w})$. In gradient descent, each iteration performs the update

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

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 $J(\mathbf{w})$.

- As we perform gradient descent, it is helpful to monitor the convergence by computing the loss, i.e., the value of the objective function J. Complete `loss_and_grad()` to calculate $J(\mathbf{w})$, and the gradient. Test your results by running the code in the main file Notebook.ipnyb. If you implement everything correctly, you should get the loss function around 4 and gradient approximately $[-3.2, -10.5]$.

We will use the following specifications for the gradient descent algorithm:

- We run the algorithm for 10,000 iterations.
- We will use a fixed step size.
- So far, you have used a default learning rate (or step size) of $\eta = 0.01$. Try different $\eta = 10^{-4}, 10^{-3}, 10^{-1}$, and make a table of the coefficients and the final value of the objective function. How do the coefficients compare?

  The coefficients were all about the same in terms of loss

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

$$\mathbf{w} = \left(\boldsymbol{X}^{\top}\boldsymbol{X}\right)^{-1}\boldsymbol{X}^{\top}\mathbf{y}$$

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

- Implement the closed-form solution `closed_form()`.
- What is the closed-form solution? How do the coefficients and the cost compare to those obtained by GD? How quickly does the algorithm run compared to GD?

  The closed form algorithm ran faster. See the code appendix for the implementation

Polynomial Regression

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

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^{\top}\phi(\mathbf{x}) = w_0 + w_1 x + w_2 x^2 + \ldots + w^m x^m$$

(f) Recall that polynomial regression can be considered as an extension of linear regression in which we replace our input matrix $\boldsymbol{X}$ with

$$\boldsymbol{\Phi} = \begin{pmatrix} \phi\left(x_1\right)^{\top} \\ \phi\left(x_2\right)^{\top} \\ \vdots \\ \phi\left(x_N\right)^{\top} \end{pmatrix}$$

where $\phi(x)$ is a function such that $\phi_j(x) = x^j$ for $j = 0, \ldots, m$.

Update `gen_poly_features()` for the case when $m \geq 2$.

See code appendix

(g) For $m = \{0, \ldots, 10\}$, use the closed-form solver to determine the best-fit polynomial regression model on the training data, and with this model, calculate the loss on both the training data and the test data. Generate a plot depicting how loss 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.

Based on the plot, the polynomial degree that best fits the data appears to be around degree 2 or 3, as indicated by the lowest point of the test error curve (blue line). Before this point, underfitting is evident, where the model is too simple to capture the underlying data pattern. After this point, overfitting occurs, as the model complexity increases and the test error starts to rise, indicating that the model is fitting to the noise rather than the underlying relationship. The training error (black line) continues to decrease with increasing complexity, which is typical as more complex models can better fit the training data, but this does not generalize well to new data, as shown by the increasing test error.

Regularization

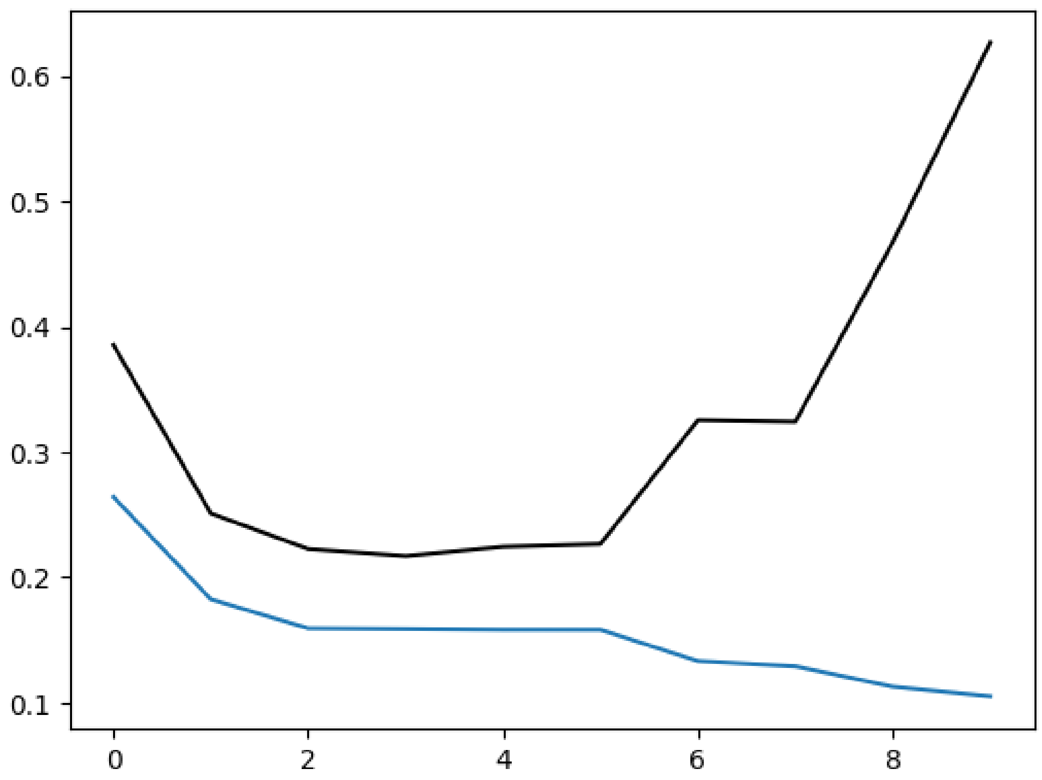

Figure 9

Finally, we will explore the role of regularization. For this problem, we will use ℓ_2 -regularization so that our regularized objective function is

$$J(\theta)=\frac{1}{2}\sum_{n=1}^{N}\left(h_\theta\left(\mathbf{x}_n\right)-y_n\right)^2+\frac{\lambda}{2}\left\|\theta_{[1:m]}\right\|^2$$

again optimizing for the parameters θ.

(h) Modify `loss_and_grad()` to incorporate ℓ_2 -regularization.

See code appendix

(i) Use your updated solver to find the coefficients that minimize the error for a tenth-degree polynomial ($m=10$) given regularization factor $\lambda=0,10^{-8},10^{-7},\ldots,10^{-1},10^{0}$. Now use these coefficients to calculate the loss (unregularized) on both the training data and test data as a function of λ. Generate a plot depicting how the loss error varies with λ (for your x-axis, let $x=[1,2,\ldots,10]$ correspond to $\lambda=\left[0,10^{-8},10^{-7},\ldots,10^{0}\right]$ so that λ is on a logistic scale, with regularization increasing as x increases). Which λ value appears to work best?

Out of all the values $\lambda=1$ appears to be the one that works best based off of the graph

Regression.py

```
import numpy as np

class Regression(object):
    def __init__(self, m=1, reg_param=0):
        self.m = m
        self.reg = reg_param
        self.dim = [m+1, 1]
        self.w = np.zeros(self.dim)

    def gen_poly_features(self, X):
        num_samples, _ = X.shape
        degree = self.m
```

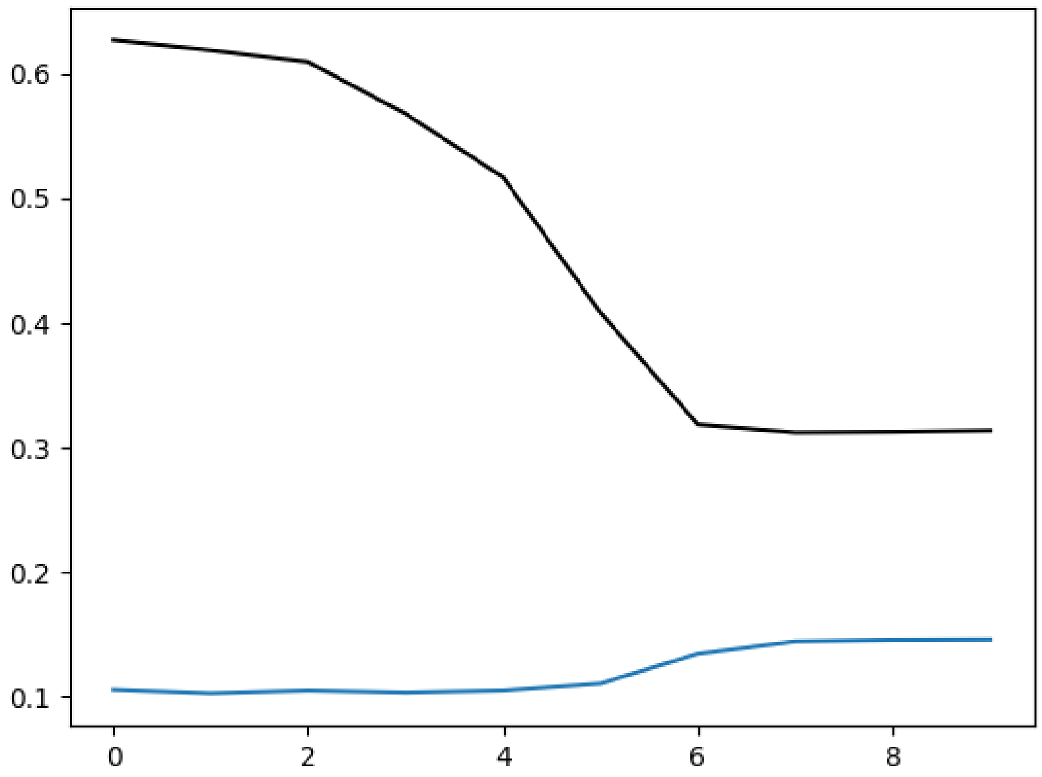

Figure 8: Loss Variation with Lambda

```
    poly_features = np.zeros((num_samples, degree+1))
    if degree == 1:
        poly_features[:, 0] = 1
        poly_features[:, 1] = X[:, 0]
    else:
        for i in range(num_samples):
            poly_features[i] = [X[i, 0]**deg for deg in range(degree + 1)]
    return poly_features

def loss_and_grad(self, X, y):
    predictions = self.predict(X)
    poly_features = self.gen_poly_features(X)
    num_samples, _ = X.shape
    loss = np.mean((predictions - y) ** 2)
    gradient = 2 * poly_features.T.dot(predictions - y) / num_samples
    if self.m > 1:
        gradient += self.reg * np.r_[0, self.w[1:]]
    return loss, gradient

def train_LR(self, X, y, eta=1e-3, batch_size=30, num_iters=1000) :
    loss_history = []
    N,d = X.shape
    for t in np.arange(num_iters):
            X_batch = None
            y_batch = None
            batch_indices = np.random.choice(N, batch_size, replace=False)
            for i in range(batch_size):
                X_batch = X[batch_indices,:]
                y_batch = y[batch_indices]
            loss = 0.0
            grad = np.zeros_like(self.w)
            self.w = self.w.flatten()
            loss, grad = self.loss_and_grad(X_batch, y_batch)
            self.w = self.w - eta*grad
            loss_history.append(loss)
    return loss_history, self.w

def closed_form(self, X, y):
    poly_features = self.gen_poly_features(X)
    self.w = np.linalg.inv(poly_features.T @ poly_features + self.reg * np.eye(poly_features.shape[1])) @ 
    loss, _ = self.loss_and_grad(X, y)
    return loss, self.w

def predict(self, X):
    poly_features = self.gen_poly_features(X)
    predictions = poly_features.dot(self.w).flatten()
    return predictions
```

Logistic.py

```
import numpy as np

class LogisticRegression(object):
    def __init__(self, num_features=784, regularization_strength=0):
        self.regularization_strength = regularization_strength
        self.weights = np.zeros((num_features + 1, 1))

    def augment_features(self, X):
        num_samples = X.shape[0]
        return np.hstack([np.ones((num_samples, 1)), X])

    def compute_loss_and_gradient(self, X, y):
        augmented_X = self.augment_features(X)
        logits = np.dot(augmented_X, self.weights).flatten()
        probabilities = 1 / (1 + np.exp(-logits))
        binary_labels = (y + 1) / 2  # Convert from {-1, 1} to {0, 1}
        loss = -np.mean(binary_labels * np.log(probabilities) + (1 - binary_labels) * np.log(1 - probabilities
        gradient = np.dot(augmented_X.T, (probabilities - binary_labels)) / num_samples
        if self.regularization_strength:
            gradient += self.regularization_strength * np.vstack([0, self.weights[1:]])
        return loss, gradient

    def train(self, X, y, learning_rate=1e-3, batch_size=1, num_iterations=1000):
        loss_history = []
        num_samples = X.shape[0]
        for _ in range(num_iterations):
            indices = np.random.choice(num_samples, batch_size, replace=False)
            X_batch = X[indices]
            y_batch = y[indices]
            loss, grad = self.compute_loss_and_gradient(X_batch, y_batch)
            self.weights -= learning_rate * grad
            loss_history.append(loss)
        return loss_history

    def predict(self, X):
        augmented_X = self.augment_features(X)
        logits = np.dot(augmented_X, self.weights)
        return np.where(logits > 0, 1, -1).flatten()
```

Notebook.ipynb

```
# %%
import numpy as np
import matplotlib.pyplot as plt
import random
import csv
from utils import mnist_reader
from utils.data_load import load
import codes
# Load matplotlib images inline
%matplotlib inline
# These are important for reloading any code you write in external .py files.
# see http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2

# %% [markdown]
# # Problem 4: Binary Classification
#
# Please follow our instructions in the same order to solve the binary classification problem.
# Please print out the entire results and codes when completed.

# %%
#X_train, y_train = mnist_reader.load_mnist('fashion-mnist/data/fashion', kind='train')
#X_test, y_test = mnist_reader.load_mnist('fashion-mnist/data/fashion', kind='t10k')

X_train = np.load('./data/binary_classification/X_train.npy')
y_train = np.load('./data/binary_classification/y_train.npy')
X_test = np.load('./data/binary_classification/X_test.npy')
y_test = np.load('./data/binary_classification/y_test.npy')

print('Train data shape: ', X_train.shape)
print('Train target shape: ', y_train.shape)
print('Test data shape: ',X_test.shape)
print('Test target shape: ',y_test.shape)

# %%
# PART (a):
# To Visualize a point in the dataset
index = 11
X = np.array(X_train[index], dtype='uint8')
X = X.reshape((28, 28))
fig = plt.figure()
plt.imshow(X, cmap='gray')
plt.show()
if y_train[index] == 1:
    label = 'Dress'
else:
    label = 'Shirt'
print('label is', label)

# %% [markdown]
# ## Train Perceptron
# In the following cells, you will build Perceptron Algorithm.

# %%
# PART (b),(c):
# Implement the perceptron Algorithm and compute the number of mis-classified point
N = X_train.shape[0] # Number of data point train
```

```
N_test = X_test.shape[0] # Number of data point test
d = X_train.shape[1] # Number of features
loss_hist = []
W = np.zeros((d+1,1))
X_train_h = np.hstack((np.ones((N,1)), X_train))
X_test_h = np.hstack((np.ones((N_test,1)), X_test))
# ================================================================== #
# YOUR CODE HERE:
# complete the following code to plot both the training and test accuracy in the same plot
# for m range from 1 to N
# ================================================================== #
test_loss_hist = []
W = W.flatten()
for i in range(N):
    x_i = X_train_h[i]
    y_i = y_train[i]
    y_pred = np.sign(np.dot(W, X_train_h[i]))
    if (y_pred != y_train[i]):
        W += y_train[i]* X_train_h[i]

        missclassified_train = 0
        missclassified_test = 0

        for j in range(N):
            train_pred = np.sign(np.dot(W, X_train_h[j]))
            if (train_pred != y_train[j]):
                missclassified_train += 1

        for k in range(N_test):
            test_pred = np.sign(np.dot(W, X_test_h[k]))
            if (test_pred != y_test[k]):
                missclassified_test += 1

        loss_hist.append(missclassified_train/N)
        test_loss_hist.append(missclassified_test/N_test)

    else:
        loss_hist.append(loss_hist[-1])
        test_loss_hist.append(test_loss_hist[-1])

# Print final losses
print(f'Final Loss Value (Train) = {loss_hist[-1]*100:.2f}%')
print(f'Final Loss Value (Test) = {test_loss_hist[-1]*100:.2f}%')
print(f'Norm of W = {np.linalg.norm(W)**2}')

# Plotting
plt.grid(True)
plt.plot(loss_hist, 'purple', label='Training Loss')
plt.plot(test_loss_hist, 'teal', label='Testing Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Training and Testing Loss Over Epochs')
plt.legend()
plt.show()

# ================================================================== #
# END YOUR CODE HERE
# ================================================================== #
# %% [markdown]
# ## Train Logistic Regression
# In the following cells, you will build a logistic regression. You will implement its loss function, then sub

# %%
from codes.Logistic import Logistic

# %%
## PART (f):
X_train = np.load('./data/binary_classification/X_train.npy')
y_train = np.load('./data/binary_classification/y_train.npy')
X_test = np.load('./data/binary_classification/X_test.npy')
y_test = np.load('./data/binary_classification/y_test.npy')
## Complete loss_and_grad function in Logistic.py file and test your results.
N,d = X_train.shape
logistic = Logistic(d=d, reg_param=0)
loss, grad = logistic.loss_and_grad(X_train,y_train)
print('Loss function=',loss)
print(np.linalg.norm(grad,ord=2)**2)

# %%
# PART (h)
# Complete predict function in Logisitc.py file and compute the percentage of mis-classified points
y_pred = logistic.predict(X_test)
test_err = np.sum((y_test!=y_pred))*100/X_test.shape[0]
print(test_err,'%')

# %%
## PART (i):
Batch = [1, 50 , 100, 200, 300]
test_err = np.zeros((len(Batch),1))
# ================================================================ #
# YOUR CODE HERE:
# Train the Logistic regression for different batch size. Average the test error over 10 times
# ================================================================ #

iters = 10
count = 0
for b in Batch:
    temp_err = 0
    for i in range(iters):
        logistic = Logistic(d = d)
        loss_hist, w = logistic.train_LR(X_train,y_train, eta=1e-5,batch_size=b, num_iters=6000/b)
        y_pred = logistic.predict(X_test)
        temp_err += np.sum((y_test!=y_pred))*100/X_test.shape[0]

    temp_err /= iters
    test_err[count] = temp_err
    count += 1

# ================================================================ #
# END YOUR CODE HERE
# ================================================================ #
fig = plt.figure()
plt.plot(Batch,test_err)
plt.xlabel('Batch_size')
plt.ylabel('Test_error')
plt.show()
fig.savefig('./plots/LR_Batch_test.pdf')

# %% [markdown]
# # Problem 5: Linear Regression
# Please follow our instructions in the same order to solve the linear regresssion problem.
#
# Please print out the entire results and codes when completed.

# %%
def get_data():
    """
    Load the dataset from disk and perform preprocessing to prepare it for the linear regression problem.
    """
    X_train, y_train = load('./data/regression/regression_train.csv')
    X_test, y_test = load('./data/regression/regression_test.csv')
    return X_train, y_train, X_test, y_test

X_train, y_train, X_test, y_test= get_data()


print('Train data shape: ', X_train.shape)
print('Train target shape: ', y_train.shape)
print('Test data shape: ',X_test.shape)
print('Test target shape: ',y_test.shape)

# %%
## PART (a):
## Plot the training and test data ##

plt.plot(X_train, y_train,'o', color='black')
plt.plot(X_test, y_test,'o', color='blue')
plt.xlabel('Train data')
plt.ylabel('Target')
plt.show()

# %% [markdown]
# ## Training Linear Regression
# In the following cells, you will build a linear regression. You will implement its loss function, then subse

# %%
from codes.Regression import Regression

# %%
## PART (c):
## Complete loss_and_grad function in Regression.py file and test your results.
regression = Regression(m=1, reg_param=0)
loss, grad = regression.loss_and_grad(X_train,y_train)
print('Loss value',loss)
print('Gradient value',grad)
##

# %%
## PART (d) (Different Learning Rates):
from numpy.linalg import norm
lrs = [1e-1, 1e-2, 1e-3, 1e-4]
test_err = np.zeros((len(lrs),1))
# ================================================================ #
# YOUR CODE HERE:
# Train the Linear regression for different learning rates and average the test error over 10 times
# ================================================================== #
lrs = [1e-1, 1e-2, 1e-3, 1e-4]
test_err = np.zeros((len(lrs),1))  # This will store the average test error for each learning rate

# Loop over each learning rate
for index, eta in enumerate(lrs):
    errors = []  # To store the test errors for averaging

    # Repeat training and testing 10 times for each learning rate
    for _ in range(10):
        regression = Regression(m=1)  # Initialize the Regression object for linear regression
        # Train the model using the current learning rate
        regression.train_LR(X_train, y_train, eta=eta, batch_size=30, num_iters=1000)

        # Predict on test data
        y_pred = regression.predict(X_test)

        # Calculate the mean squared error for the test data
        error = np.mean((y_test - y_pred) ** 2)
        errors.append(error)

    # Average the test errors and store them
    test_err[index] = np.mean(errors)
# ================================================================== #
# END YOUR CODE HERE
# ================================================================== #
fig = plt.figure()
plt.plot(lrs,test_err)
plt.xlabel('Learning Rate')
plt.ylabel('Test_error')
plt.show()
fig.savefig('./plots/LR_Batch_test.pdf')

# %%
## PART (e):
## Complete closed_form function in Regression.py file
loss_2, w_2 = regression.closed_form(X_train, y_train)
print('Optimal solution loss',loss_2)
print('Optimal solution gradient',w_2)

# %%
## PART (g):
train_loss=np.zeros((10,1))
test_loss=np.zeros((10,1))
# ================================================================== #
# YOUR CODE HERE:
# complete the following code to plot both the training and test loss in the same plot
# for m range from 1 to 10
# ================================================================== #
for m in range(1,11):
    regression = Regression(m=m, reg_param=0)
    loss, w = regression.closed_form(X_train, y_train)
    train_loss[m-1] = loss
    y_pred = regression.predict(X_test)
    temp_loss = 0
    for i in range(X_test.shape[0]):
        temp_loss += (y_pred[i]-y_test[i])**2
    temp_loss /= X_test.shape[0]
    test_loss[m-1] = temp_loss

# ================================================================== #
# END YOUR CODE HERE
# ================================================================== #
plt.plot(train_loss)
plt.plot(test_loss, color='black')
plt.show()

# %%
#PART (i):
train_loss=np.zeros((10,1))
test_loss=np.zeros((10,1))
# ================================================================== #
# YOUR CODE HERE:
# complete the following code to plot both the training and test loss in the same plot
# for m range from 1 to 10
# ================================================================== #
lambdas = [0, 1e-8, 1e-7, 1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1e0]
count = 0
for la in lambdas:
    regression = Regression(m=10, reg_param=la)
    loss, w = regression.closed_form(X_train, y_train)
    y_pred = regression.predict(X_test)
    temp_loss = 0
    for i in range(X_test.shape[0]):
        temp_loss += (y_pred[i]-y_test[i])**2
    temp_loss /= X_test.shape[0]
    train_loss[count] = loss
    test_loss[count] = temp_loss
    count += 1
# ================================================================== #
# END YOUR CODE HERE
# ================================================================== #

plt.plot(train_loss)
plt.plot(test_loss, color='black')
plt.show()
```