CS 240 : Lab 5
Perceptron Classifiers

TAs: Vidit Goel, Rounak Dalmia

Instructions

- This lab will be **graded**. The weightage of each question is provided in this PDF.
- Please read the problem statement and the submission guidelines carefully.
- All code fragments need to be written within the `TODO` blocks in the given Python files. Do not change any other part of the code.
- **Do not** add any **additional** *print* statements to the final submission since the submission will be evaluated automatically.
- For any doubts or questions, please contact either the TA assigned to your lab group or one of the 2 TAs involved in making the lab.
- The deadline for this lab is **Monday, 12 February, 5 PM**.
- The submissions will be checked for plagiarism, and any form of cheating will be appropriately penalized.

The submissions will be on Gradescope. You need to upload the following Python files: `q1.py` and `q2.py`. In Gradescope, you can directly submit these Python files using the upload option (you can either drag and drop or upload using browse). No need to create a tar or zip file.

1 Perceptron Classifier [60 marks]

In statistics and machine learning, *classification* refers to a type of supervised learning. For this task, training data with known class labels are given and used to develop a classification rule for assigning new unlabeled data to one of the classes. A special case of the task is binary classification, which involves only two classes. Some examples:

- Classifying an email as spam or non-spam.
- Classifying a tumor as benign or malignant.

In the last lab, we talked about logistic regression. In this lab, we will implement another famous binary classifier, perceptron. It consists of a single node or neuron that takes a row of data as input and predicts a class label. This is achieved by calculating the weighted sum of the inputs and a bias (set to 1). The weighted sum of the input of the model is called the activation.

$$\textbf{Activation} = \text{Weights} \times \text{Inputs} + \text{Bias}$$

If the activation is above 0.0, the model will output 1.0; otherwise, it will output 0.0.

$$\text{Predict } \textbf{1}\text{: If } \textbf{Activation } > 0.0$$
$$\text{Predict } \textbf{-1}\text{: If } \textbf{Activation } \leq 0.0$$

Many advanced libraries, such as `scikit-learn`, make it possible for us to train various models on labeled training data, and predict on unlabeled test data, with a few lines of code. However, it does not give an insight into the details of what happens underneath when we run those codes. In this code, we implement a perceptron classifier from scratch, without using any advanced libraries, to understand how it works in the context of binary classification. **You cannot import any other libraries except the one you are already provided with.** The basic idea is to segment the computations into pieces and write functions to compute each piece sequentially to build a function based on the previously defined functions.

1.1 Problem Statement and Dataset

Recall, the Perceptron weight update rule for a *misclassified* example (x, y) is

$$w_{t+1} \leftarrow w_t + \eta y x$$

where $w \in \mathbb{R}^3$ and η is a learning rate hyperparameter (that we set as 1 in class). Append a 1 at the end of each x to allow for the bias term in w to be computed. Implement the function `fit` inside the class `Perceptron` and do the following for a fixed number of epochs set in `epochs`. Note this is different from what we studied in class. If w_t is the weight vector at the start of epoch number t, then compute the predictions for all training examples using w_t and compute the sum of $\eta y_i x_i$ across all x_i's that result in a misclassification using w_t. If the latter sum is saved in Δ, then the updated weight vector will be:

$$w_{t+1} \leftarrow w_t + \Delta$$

This weight update can be done in a single line of code. The vanilla Perceptron classifier then returns the final weight vector after a predefined number of epochs.

Exponential moving average (EMA)-variant of the Perceptron: A variant of the Perceptron update rule is where an averaged weight vector is computed in every epoch, instead of a complete update based on one data point. We suggest a variant that keeps track of the exponential moving average of the weight vector:

$$a_{t+1} = \lambda a_t + (1 - \lambda) w_{t+1}$$

where w_t is the weight vector at epoch t of the training and $a_0 = w_0$. The parameter λ controls how quickly the moving average reacts to the changes in w. Higher the value of λ, the greater its inertia will be. The exponential moving average can be used in place of the final weight vector to perform the final prediction on the test set. This is seen to give more stable results, as illustrated by our task. Implement both the update for the moving average and the prediction function within `fit`. The `plot_decision_boundary` method stores images of the linear decision boundary changing over time for both the final weight and moving average-based predictions. You can then run `q1/make_gif.py` to make an animation out of these images and see the decision boundaries change over time. Note that this is an **ungraded exercise**. Notice how the decision boundary is much more stable when the average is used. Complete all the TODOs marked in `q1/q1.py`. You should also print the test accuracy using both the final weight vector and the averaged weight vector.

1.2 Tasks to be completed

We have provided a `data.csv` file, which consists of a sample dataset. We have also divided the dataset into train and test splits. In the file `q1.py`, you are expected to:

- Task 1: Complete the `fit` function in the `Perceptron` class using the algorithm as described in the problem statement above. Ideally, the weight-update step can be implemented in a single line. [30 marks]
- Task 2: Complete the `predict` function, which is essentially calling the weights learned by the Perceptron classifier and returning the predicted labels. [10 marks]
- Task 3: Compute the **average accuracy**, on the test dataset, which is basically the number of correctly classified samples divided by total samples. Compute this accuracy for both the vanilla classifier and the moving-average classifier. [10+10 marks]

To run the file, simply use the command line argument: `python3 -W ignore q1.py --dataset data1.csv`

1.3 Testing

The following components of your code will be tested. Make sure that the variables returned by these functions (and others in general) follow the prescribed format.

- The `Perceptron.fit` function
- The `Perceptron.predict` function
- Average accuracy

Note, for your verification purpose, the expected output for the given dataset is given in the file `expected_out.txt`

2 Stochastic Gradient Descent using Hinge Loss [40 marks]

Hinge Loss is a loss function widely used in binary classification tasks. It is defined as follows:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n} \max(0, 1 - y_i \cdot (\mathbf{x}_i \cdot \mathbf{w}^T))$$

where $\mathbf{x}_i \in \mathbb{R}^{(d+1)}$, $y_i \in \mathbb{R}$, for all $i = 1, \ldots, n$, and $\mathbf{w} \in \mathbb{R}^{(d+1)}$. For a single data point $\mathbf{x}_i$, this loss essentially takes the following form:

$$\mathcal{L}_i(\mathbf{w}) = \max(0, 1 - y_i \cdot (\mathbf{x}_i \cdot \mathbf{w}^T))$$

For this problem, the training set consists of

- dataset $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$,
- with $\mathbf{x}_i$ being a 3-dimensional vector $\mathbf{x}_i = (x_i^{(1)}, x_i^{(2)}, 1)$,
- the label y_i of $\mathbf{x}_i$ can take K possible values, e.g., with $K = 2$ classes, $y^{(i)} \in \{-1, 1\}$.

The task is similar to the previous question, but we explore a slightly different method to learn the weights and classify the dataset. For this problem, you are required to minimise the hinge loss and optimise the weights using a variant of the stochastic gradient descent algorithm discussed in lectures. We randomly sample a data point from the training dataset available to us and update the weights using the hinge loss gradient at that point. Note that while sampling these points randomly, we may generate skewed distributions, but if we sample a large number of time, we can expect to get a uniform distribution. Note that in the problem statement, we have given the formula for hinge loss but you will require **gradient** of hinge loss. You should calculate the gradient as an exercise.

Algorithm 1 Stochastic Gradient Descent

$w \leftarrow zero\_vector()$
$w\_values \leftarrow [w]$
$step \leftarrow 0$
for *epoch in range*(*epochs*) **do**
  for i *in range*(n) **do**
    $index \leftarrow random(0, n-1)$
    $dw \leftarrow \nabla f(x[index], y[index], w)$ ▷ $x[index], y[index]$ represents x, y values of $index$-th datapoint
    $w = w - lr * dw$
    append w to w_values
    $step \leftarrow step + 1$
  end for
end for
return w_values

2.1 Code Tasks and Functions

Same as above, we have provided a `data.csv` file, which consists of a sample dataset. We have also divided the dataset into train and test splits. In the file `q2.py`, you are expected to:

- Task 1: Complete the helper function `hinge_loss_gradient` that will be used in the stochastic gradient step.
- Task 2: Complete the `fit` function in the `Perceptron` class using the algorithm as described in the problem statement above. [20 marks]
- Task 3: Complete the `predict` function, which is essentially calling the weights learned by the Perceptron classifier and returning the predicted labels. This part is the same as in the first question. [10 marks]
- Task 4: Compute the **average accuracy** on the test dataset, which is basically the number of correctly classified samples divided by total samples. [10 marks]

To run the file, simply use the command line argument: `python3 -W ignore q2.py --dataset data1.csv`

2.2 Testing

The following components of your code will be tested. Make sure that the variables returned by these functions (and others in general) follow the prescribed format.

- Correctness of `hinge_loss_gradient` will be tested solely on the basis of gradient descent.
- The `Perceptron.fit` function
- The `Perceptron.predict` function
- Average accuracy

Note, for your verification purpose, the expected output for the given dataset is given in the file `expected_out.txt`