George Mason University

Learning From Data Fall 2024

Computer Exercise #2

Assigned: September 04, 2024
Due Date: September 12, 2024

This computer exercise is concerned with the perceptron learning algorithm. Most of the code given in this assignment is available in a Jupyter notebook that is distributed along with this exercise.

The due date for this exercise is September 12, 2024 at midnight. Absolutely no late submissions will be accepted after this date. It is important for you to finish this assignment on time and move on to the next. If you have not finished, submit the work you have done for partial credit.

Review the introduction to the first computer exercise to see what the purpose of this and all future computer exercises is. In this assignment, there are a number of exercises and experiments to run to help you understand PLA. Do not view this assignment as tightly scripted - feel free to explore new ideas and perform additional experiments. A few ideas are given in the assignement.

A second Jupyter notebook, `cset_2_report.ipynb` is provided for the submission of your write-up. As in the first exercise, it is not necessary to duplicate the code given to you in this report, but any new code or code you feel is important to describe or document your work is welcome. In addition, only provide plots and graphs that are important. Do not pad your report with code, figures, and plots that do not have any value.

Notes

1. In your reports you will probably want to include some plots/figures. To avoid having to create a code cell, insert the code and run it to display the plot, in your working Jupyter notebook you can save a plot as a `jpg` image and then import it into your report. This will significantly shorten the amount of material in your submitted report and make it easier to read and grade. To save a plot as a `jpg` image, do the following:

   ```
   plt.savefig('figure_name.jpg')
   ```

   You can save it other formats such as `png` as well. There are then several ways to insert the saved image into your notebook.

   (a) In a code cell, import the `image` class

   ```
   from IPython.display import Image
   ```

   and then display the image as follows:

   ```
   Image('figure_name.jpg',width=300,height=300)
   ```

   This approach allows you to set the size of the image. to be displayed.

   (b) In a markdown cell you can copy the image and paste it in the cell with a simple `Ctrl+C` and `Ctrl+V`

   (c) Finally, you can simply drag and drop the image file into a Markdown cell.

2. Note that each time you create a dataset, it will be diffent unless you set `random_state` to some number.

3. It is important to restart your kernel when things do not look right or when you make significant changes to your code, as some parameters or variables set on previous runs or in other cells may affect your results.

Computer Exercise 2.1 (The Perceptron Learning Algorithm):

The Perceptron Learning Algorithm (PLA) is a simple classification algorithm suitable for large scale learning. It uses stochastic gradient descent (SGD) to find a **linear classifier** that minimizes the **perceptron error**,

$$R_N(g) = \sum_{k=1}^{N} |\boldsymbol{w}^T \boldsymbol{x}_k| I(y_k \neq g(\boldsymbol{x}_k)) = \sum_{k \in \mathcal{M}} |\boldsymbol{w}^T \boldsymbol{x}_k|$$

where $\mathcal{M}$ is the set of samples in the training set that are misclassified by $\boldsymbol{w}$. Below is a simple python program for the perceptron algorithm.

```
def perceptron_sgd(X, y):
    w = np.zeros(len(X[0]))
    eta = 1
    epochs = 20
    for t in range(epochs):
        for i, x in enumerate(X):
            if (np.dot(X[i], w)*y[i]) <= 0:
                w = w + eta*X[i]*y[i]
    return w
```

where the training samples are stored in `X` and the target values in `y`. In this program, the step size is `eta=1` and PLA is run for 20 epochs.

If the training set is linearly separable, then PLA will converge in a finite number of iterations to a solution that correctly classifies all samples. If the data is not linearly separable, then PLA will never converge, and the solution that it returns after a termination condition has been satisfied, such as the maximum number of iterations, may not be the best one that it has found throughout the training process. One way to improve PLA, but at a cost, is the **pocket algorithm**, which is a simple way to ensure that, of all solutions evaluated with PLA, the best one is reported at the end. The idea is very simple. Set $\boldsymbol{w}_p = \boldsymbol{w}_0$, where $\boldsymbol{w}_p$ is the pocket algorithm model and $\boldsymbol{w}_0$ is the initial model used to start PLA. Find the perceptron error on the training set, $R_N(\boldsymbol{w}_0)$, and set $R_N(\boldsymbol{w}_p)$ equal to $R_N(\boldsymbol{w}_0)$. After each PLA update to a new model, $\boldsymbol{w}_n$, the perceptron error is found, and if $R_N(\boldsymbol{w}_n) < R_N(\boldsymbol{w}_p)$ then the pocket algorithm model is updated to $\boldsymbol{w}_n$ and $R_N(\boldsymbol{w}_p)$ is replaced with $R_N(\boldsymbol{w}_n)$. When the stopping criterion is satisfied, the solution $\boldsymbol{w}_p$ is reported along with the error $R_N(\boldsymbol{w}_p)$.

If you can, modify the code for the perceptron algorithm above to implement the pocket algorithm. To do this, you will need to write an **evaluation function** to find the perceptron error on the training set, and modify the perceptron code to keep track of the best classifier. Save your pocket algorithm so that you can compare its performance to PLA on datasets that are not linearly separable.

(a) **The Perceptron Learning Algorithm in Scikit-*Learn***

The perceptron algorithm can be found in the `linear_model` library of Scikit-*Learn* and may be instantiated as follows:

```
from sklearn.linear_model import Perceptron
clf = Perceptron()
```

The `Perceptron` class uses Stochastic Gradient Descent (SGD) to learn a classifier $g(\boldsymbol{x})$ that minimizes the perceptron error. There are some parameters that may be important to set instead of accepting the default values, and it would be useful to read the documentation to learn more about them. Some of the parameters that are of particular interest are:

- `tol, default=1e-3`: The iterations will stop when

```
loss > previous_loss - tol
```

- `max_iter, int, default=1000`: the maximum number of passes over the training data, i.e., the number of epochs.
- `shuffle, default=True`: If `True` then the training data is shuffled after each epoch.
- `eta, default=1`: Constant by which the updates are multiplied, i.e., the SGD step size, sometimes called the learning rate parameter.

Once the `Perceptron` class has been instantiated, the `fit` method may used to find a linear model for a dataset `(X,y)`

```
clf.fit(X,y)
```

There are some attributes of the `Perceptron` class that may be of interest, including the model parameters, and the number of iterations of PLA.

- `coef_`: The weight vector $\boldsymbol{w}$ of the linear classifier.
- `intercept_`: The bias (constant) of the classifier.
- `n_iter_`: The number of iterations before PLA was terminated, due to convergence or reaching a stopping criterion.

After a linear model has been learned, some useful methods include `score` and `predict`.

- `score(X,y)`: Returns the mean accuracy for the labeled data `(X,y)`.
- `predict(X)`: Predicts class labels for samples in X.

(b) **PLA on Some Data Sets**

To gain some experience with PLA and understand its properties, the best thing to do is to use it on some datasets where the desired discriminant function is known. In any machine learning application, before any trainning is done, it is important to create a test set to evaluate the performance of a classifier on new, unseen data. The training error is not a good measure of how well a classifier will work on new data because it is the one that is found to minimize the error on the training set, and it may not necessarily work as well on other datasets. In other words, it is *tuned* to the training set.

Given a dataset `(X,y)`, the function `train_test_split` in the `model_selection` class may be used to split the data into a training set and a test set as follows,

```
from sklearn.model_selection import train_test_split
X_train,X_test,y_train,y_test = train_test_split(X,y,test_size=0.25)
```

The size of the test set, specified by `test_size`, may be given either in terms of the number of samples, `test_size=500`, or a percentage of the total number of samples, such as `test_size=0.25`. The remaining samples will be in the training set Instead of setting the number of samples in the test set, the size of the training set may be specified by assigning a value to the parameter `train_size`., It is also possible to set a value for both `test_size` and `train_size`,

```
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.3,train_size=0.5)
```

but they should sum to a value that does not exceed the total size of the dataset.

Some sample datasets are given below. Only the first dataset is linearly separable.

Data Set 1

The first dataset consists of 100 two-dimensional features, $\boldsymbol{X} = [x_1, x_2]^T$ that are uniformly distributed over the unit square:[1]

$$0 \leq x_1, x_2 \leq 1$$

This random set may be generated as follows:

```
import numpy as np
X1=np.random.rand(n,d)
```

where `n` is the number of random samples and `d` is the dimension. Each sample is then assigned a label according to the following rule,

$$y = \text{sgn}\{g(\boldsymbol{x})\}$$

where

$$g(\boldsymbol{x}) = -0.5x_1 + x_2 - 0.25$$

This assignment may be done using the statement

```
y1=(-0.5*X1[:,0]+X1[:,1]-0.25>0).astype(int)
```

We now have a **linearly separable** dataset, `(X1,y1)`, where `X1` is an array of size 100×2 and `y1` is a vector of length 100.

Question: Will PLA work with target values of $y = 0$ and 1, or is it necessary to convert them to $y = \pm 1$? Does the `Perceptron` class care what the target values are?

Data Set 2

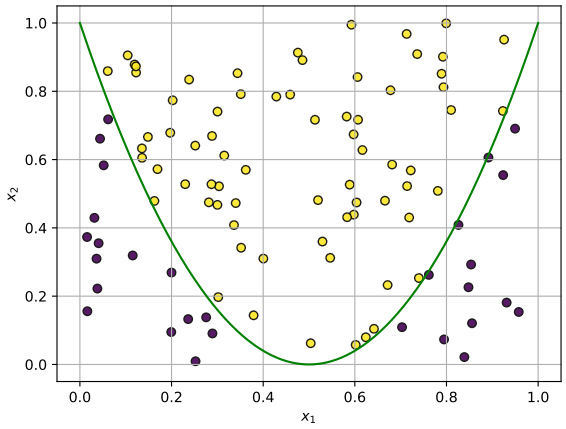
As before, create a set of random samples over the square $0 \leq x_1, x_2 \leq 1$, and let

$$g(\boldsymbol{x}) = x_2 - 4(x_1 - 0.5)^2 = 0$$

be the discriminant function that is to be learned.[2] The target values are equal to $+1$ for values of $\boldsymbol{x}$ for which $g(\boldsymbol{x}) > 0$, and are equal to zero otherwise. This dataset may be generated as follows:

```
X2=np.random.rand(100,2)
y2 = (X2[:,1] > 4*(X2[:,0]-0.5)**2).astype(int)
```

Unlike the first dataset, this dataset is not linearly separable. An example of what this data set might look like is shown in the following figure along with the discriminant function.

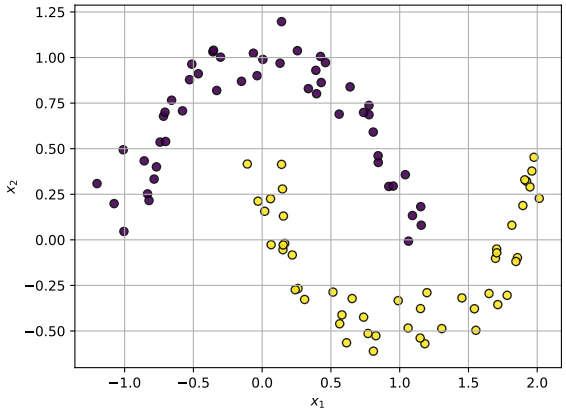


[1] Note that the first feature in a feature vector is denoted by x_1, but in python, the first element is `x[0]`. This can sometimes be confusing, so it is important to keep this difference in mind.

[2] Note: Feel free to consider ofther nonlinear discriminant functions such as $g(\boldsymbol{x}) = x_2 - \sin(\pi x_1/2)$.

Data Set 3

The next dataset, again not linearly separable, is the moon dataset consisting of a pair of interlocking half circles. An example of what this data set might look like is shown below.



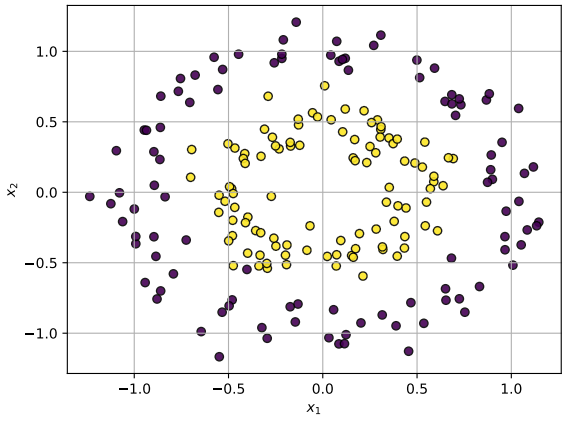
To create this dataset, use the `make_moons` class in the `sklearn.datasets` library,

```
from sklearn.datasets import make_moons
X3, y3 = make_moons()
```

This dataset is created by randomly placing points around two half circles, and then adding Gaussian noise to each point. The two parameters that need to be defined are the number of data samples to generate, `n_samples` (default value of 100), and the standard deviation of the Gaussian noise, `noise`, that has a default value `noise=None`. For the plot above, `n_samples=100` and `noise=0.1`.

Data Set 4

Although a linear perceptron is not a total failure on the moons data set, it will completely fail on the circles dataset. An example of one of these datasets is shown in the following figure.



The `make_circles` class may be imported from the `dataset` library,

```
from sklearn.datasets import make_circles
X4,y4=make_circles(noise=0.1,factor=0.5)
```

Similar to the moons dataset, `make_circles` creates a random set of points around two circles, a smaller circle inside a larger circle. The separation between the two circles is controlled by the

parameter `factor` and the number of points that are generated is specified by the parameter `n_samples`, with the default being 100. If a single number is given, then half of the points will be generated around each circle. If a two-element tuple is given, then each number specifies the number of points around each circle. The amount of noise added to the points around each circle is specified by the parameter `noise`.

PLA Experiments

For each of the four datasets above,

1. Separate the dataset into a training set and a test set, and use the training set to learn a classifier using PLA.
2. Does PLA converge? If so, how many epochs were necessary and what is the accuracy of the classifier on the test set? If it did not converge,
   a. How many epochs were run, and what stopping criterion was used? Pay attention to your choice of `tol` as this will affect when PLA stops.
   b. What is the accuracy of the classifier on the training and test sets?
   b. If you have a working pocket algorithm, use it to learn a classifier and compare its performance to PLA.
3. Run PLA using different random number seeds to see what difference, if any, this makes on the design.
4. To get a feeling of how well PLA performed, make a scatter plot of your test set along with the discriminant function that is learned.

Extra Credit

1. Make a plot of the learning curve, i.e., the classification error rate versus epoch. Discuss what you learn from these plots.
2. For data set #1, the classifier

$$g^*(\boldsymbol{x}) = x_2 - 0.5x_1 - 0.25$$

has zero expected risk,

$$R(g^*) = E\{c(\boldsymbol{x}, y, g^*(\boldsymbol{x})\} = 0$$

Can you find the expected risk of the classifier $g(\boldsymbol{x})$ that is learned using PLA?

Hint: The *shoelace theorem* may be of some help.

```
def shoelace_area(x1, y1, x2, y2, x3, y3):
    # Calculate the area using the Shoelace formula
    area = abs(x1*y2+x2*y3+x3*y1-(y1*x2+y* x3+y3*x1))/2
    return area

# Example usage
x1, y1 = 2, 1
x2, y2 = 4, 5
x3, y3 = 7, 8

area = shoelace_area(x1, y1, x2, y2, x3, y3)
print(f"The area of the triangle is: {area} square units")
```

Computer Exercise 2.2 (Nonlinear Perceptron Learning Algorithm):

Since datasets 2-4 are not linearly separable, to get a better-performing classifier it is necessary to use a nonlinear classifier. One way to do this is to add additional features that are nonlinear functions of x_1 and x_2. For example, to design a classifier using a discriminant function that is a second order polynomial, the feature vector $\boldsymbol{x}$ would be augmented with three additional features as follows:

$$\boldsymbol{z} = [x_0,\ x_1,\ x_2,\ x_1^2,\ x_1x_2,\ x_2^2]^T$$

Learning a linear classifier using the feature vector $\boldsymbol{z}$

$$g(\boldsymbol{z}) = w_0 + \sum_{k=1}^{5} w_k z_k$$

is equivalent to designing a **nonlinear classifier in x**,

$$g(\boldsymbol{x}) = w_0 + w_1x_1 + w_2x_2 + w_3x_1^2 + w_4x_1x_2 + w_5x_2^2$$

A feature vector wth quadratic features may created as follows.

```
from sklearn.preprocessing import PolynomialFeatures
poly = PolynomialFeatures(degree=2, include_bias=True)
X_train_aug = poly.fit_transform(X_train)
```

Higher order polynomials may be created by defining a different value for `degree`.

Nonlinear PLA Experiments
Repeat the experiments you did in Computer Exercise #2.1 and use a nonlinear perceptron algorithm to learn a nonlinear classifier. Experiment with different types of nonlinearities. You may also want to see what happens if you use a highly nonlinear perceptron on linearly separable data.