Perceptron

September 5, 2024

1 Perceptron Manual Implementation

1.0.1 Jack DeGesero

In this project, I manually implement the machine learning classifier known as perceptron. The goal of this model is to draw an n-dimensional boundary between all positive and negative data instances. This boundary will allow us to predict if a new instance is positive or negative dependent on its location to the boundary with some degree of accuracy. The primary hyperparameter of the model is the number of training iterations. The more iterations the model endures, the more refined the boundary becomes. See all source code below.

1.1 Data Preprocessing

Here I import the packages used for analysis and the data. Then, I will partition the data into a 70/30 split. 70% of the data will be used for training the model, the last 30% will test the accuracy of the model. I will also cleanup the data before analysis (e.g. change numerical values and drop missing values).

[1]:
```
import numpy as py #for series
import pandas as pd #for dataframe manipulation
```

[2]:
```
#read csv as a dataframe, drop all missing values
df = pd.read_csv("Single Layer Perceptron Dataset.csv").iloc[:,1:].dropna()
```

[3]:
```
df #preview
```

[3]:

	Feature1	Feature2	Feature3	Class_Label
0	1.0	0.08	0.72	1.0
1	1.0	0.10	1.00	1.0
2	1.0	0.26	0.58	1.0
3	1.0	0.35	0.95	0.0
4	1.0	0.45	0.15	1.0
5	1.0	0.60	0.30	1.0
6	1.0	0.70	0.65	0.0
7	1.0	0.92	0.45	0.0
8	1.0	0.42	0.85	0.0
9	1.0	0.65	0.55	0.0
10	1.0	0.20	0.30	1.0
11	1.0	0.20	1.00	0.0

```
12       1.0       0.85       0.10           1.0
```

[4]:
```
#70/30 test-train split

trainRows = int(df.shape[0] * .7) #value of where to split data (index 70%)

#slice dataframe
xtrain, xtest, ytrain, ytest = df.iloc[0:trainRows].iloc[:,:3],df.
 ↪iloc[trainRows:].iloc[:,:3],df.iloc[0:trainRows].iloc[:,3:],df.
 ↪iloc[trainRows:].iloc[:,3:]
```

[5]:
```
xtrain#preview of pandas dataframe
```

[5]:
```
    Feature1  Feature2  Feature3
0        1.0      0.08      0.72
1        1.0      0.10      1.00
2        1.0      0.26      0.58
3        1.0      0.35      0.95
4        1.0      0.45      0.15
5        1.0      0.60      0.30
6        1.0      0.70      0.65
7        1.0      0.92      0.45
8        1.0      0.42      0.85
```

[20]:
```
ytrain.replace(0.0, -1.0, inplace=True) #replace 0 with -1
ytrain#preview of pandas dataframe
```

[20]:
```
    Class_Label
0           1.0
1           1.0
2           1.0
3          -1.0
4           1.0
5           1.0
6          -1.0
7          -1.0
8          -1.0
```

[106]:
```
xtest#preview of pandas dataframe
```

[106]:
```
     Feature1  Feature2  Feature3
9        1.0      0.65      0.55
10       1.0      0.20      0.30
11       1.0      0.20      1.00
12       1.0      0.85      0.10
```

[69]:
```
ytest.replace(0.0, -1.0, inplace=True) #replace instances of zero with -1
ytest#preview of pandas dataframe
```

[69]:
```
    Class_Label
9          -1.0
10          1.0
11         -1.0
12          1.0
```

1.2 Model Training

Here the model will be trained by the function 'perceptronTrain'. It accepts 3 parameters: the training rows X, the training rows Y, and the hyperparameter, # of iterations. The model, when complete, will return the boundary respresented as a weight vector and a bias.

[124]:
```
def perceptronTrain(trainX, trainY, maxIter):

    w = py.zeros(len(trainX.columns))#weight vector

    b=0.0#bias

    for h in range(maxIter):
        for i in range(trainX.shape[0]):
            x = trainX.iloc[i].values #cur x tuple
            y = trainY.iloc[i].item() #cur y value, used for conditional␣
 ↪updating

            activation = py.dot(x,w)+b

            if y * activation <= 0:
                for k in range(len(w)):
                    w[k] += y * x[k]
                b += y

    return w,b
```

1.3 Model Testing

After training the model, it maps the testing instances and records where they lie respective to the boundary. Which side of the boundary an instance is mapped to will determine the predicted value. The predicted values are then recorded and compared to the actual values to check how many it got right, measured by accuracy. This function, perceptronTest, will return the accuracy of the boundary and the predictions it has made.

[156]:
```
def perceptronTest(testX, testY, w, b):

    predicts = []

    for i in range(testX.shape[0]):
        x = testX.iloc[i].values
```

```
        activation = py.dot(x, w) + b

        if activation > 0:
            predict = 1
        else:
            predict = -1

        predicts.append(predict)

    predicts = py.array(predicts)

    return predicts
```

1.4 Evaluation

Here we will run the model in 7 different tests, each one consisting of a different number of max iterations (1,5,10,20,30,40,50). Each test will compute a different vector of predictions. We then compare the predicted values of each test to the vector of actual values to assess accuracy.

[198]:
```
print("Original Y values for test:", ytest.values[:].ravel())
for x in [1,5,10,20,30,40,50]:
    weight, bias = perceptronTrain(xtrain, ytrain, maxIter=x)
    preds = perceptronTest(xtest,ytest, weight, bias)
    print("Accuracy of", x, "training iteration(s):", (ytest.values[:].ravel()␣
 ↪== preds).mean() , "The predicted values:", preds)
```

```
Original Y values for test: [-1.  1. -1.  1.]
Accuracy of 1 training iteration(s): 0.5 The predicted values: [-1 -1 -1 -1]
Accuracy of 5 training iteration(s): 1.0 The predicted values: [-1  1 -1  1]
Accuracy of 10 training iteration(s): 0.75 The predicted values: [-1  1 -1 -1]
Accuracy of 20 training iteration(s): 0.5 The predicted values: [1 1 1 1]
Accuracy of 30 training iteration(s): 0.5 The predicted values: [1 1 1 1]
Accuracy of 40 training iteration(s): 0.5 The predicted values: [1 1 1 1]
Accuracy of 50 training iteration(s): 0.5 The predicted values: [1 1 1 1]
```

1.5 Results

We can see that the boundary at 1 iteration has a weak prediction accuracy of 50% (only classified 2 correctly). As the model reaches 5 iterations, it correctly classifies all tuples (best case). At 10 iterations, it guesses 3 correctly. Everything after 10 iterations: 20,30,40, and 50, all have an accuracy of 50%. These results provide us insight into the importance of tweaking hyperparameters to increase accuracy. By selecting the right hyperparameters, we can generalize the model to accurately classify new instances of data.