National University of Computer & Emerging Sciences

Lab Manual

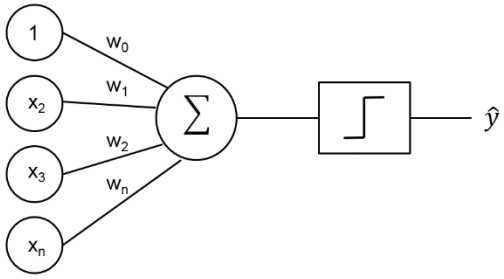
CS461: Artificial Intelligence Lab

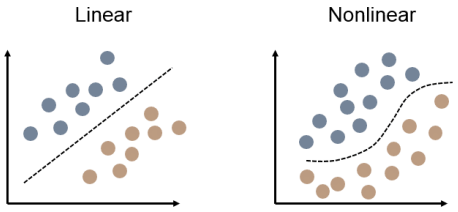
Course Instructor	Dr. Hafeez-Ur-Rehman
Lab Instructor	Muhammad Yousaf
Semester	Spring 2021

Perceptron

Introduction

The Perceptron algorithm is the simplest type of artificial neural network.
It is a model of a single neuron that can be used for two-class classification problems and provides the foundation for later developing much larger networks.





Perceptron Algorithm

The Perceptron is inspired by the information processing of a single neural cell called a neuron.
A neuron accepts input signals via its dendrites, which pass the electrical signal down to the cell body. In a similar way, the Perceptron receives input signals from examples of training data that we weight and combined in a linear equation called the activation.

activation = sum (weight_i * x_i) + bias

The activation is then transformed into an output value or prediction using a transfer function, such as the step transfer function.

prediction = 1.0 if activation >= 0.0 else 0.0

The weights of the Perceptron algorithm must be estimated from your training data using stochastic gradient descent.

Stochastic Gradient Descent

Gradient Descent is the process of minimizing the **error** by following the gradients of the cost function. In machine learning, we can use a technique that evaluates and updates the weights every iteration called stochastic gradient descent to minimize the error of a model on our training data.

The way this optimization algorithm works is that each training instance is shown to the model one at a time. The model makes a prediction for a training instance, the error is calculated and the model is updated in order to reduce the error for the next prediction. This procedure can be used to find the set of weights in a model that result in the smallest error for the model on the training data.

For the Perceptron algorithm, each iteration the weights (w) are updated using the equation:

w = w + learning_rate * (expected - predicted) * x

Where w is weight being optimized, learning_rate is a learning rate that you must configure (e.g. 0.01), (expected – predicted) is the prediction error for the model on the training data attributed to the weight and x is the input value.

Algorithm

This tutorial is broken down into 3 parts:

1. Making Predictions.
2. Training Network Weights.

These steps will give you the foundation to implement and apply the Perceptron algorithm to your own classification predictive modeling problems.

Making Predictions

Develop a function that can make predictions.
Below is a function named **predict ()** that predicts an output value for a row given a set of weights.
The first weight is always the bias as it is standalone and not responsible for a specific input value.

```
# Make a prediction with weights
def predict(row, weights):
    activation = weights[0]
    for i in range(len(row)-1):
        activation += weights[i + 1] * row[i]
    return 1.0 if activation >= 0.0 else 0.0
```

Dataset

X1	X2	Y
2.7810836	2.550537003	0
1.465489372	2.362125076	0
3.396561688	4.400293529	0
1.38807019	1.850220317	0
3.06407232	3.005305973	0
7.627531214	2.759262235	1
5.332441248	2.088626775	1
6.922596716	1.77106367	1
8.675418651	-0.242068655	1
7.673756466	3.508563011	1

We can also use previously prepared weights to make predictions for this dataset.
Putting this all together we can test our **predict()** function below.

```
weights = [-0.1, 0.20653640140000007, -0.23418117710000003]
for row in dataset:
    prediction = predict(row, weights)
    print("Expected=%d, Predicted=%d" % (row[-1], prediction))
```

Training Network Weights

We can estimate the weight values for our training data using stochastic gradient descent.

Stochastic gradient descent requires two parameters:

Learning Rate: Used to limit the amount each weight is corrected each time it is updated.
Epochs: The number of times to run through the training data while updating the weight.
These, along with the training data will be the arguments to the function.

There are 3 loops we need to perform in the function:

1. Loop over each epoch.
2. Loop over each row in the training data for an epoch.
3. Loop over each weight and update it for a row in an epoch.

As you can see, we update each weight for each row in the training data, each epoch.

Weights are updated based on the error the model made. The error is calculated as the difference between the expected output value and the prediction made with the candidate weights.

There is one weight for each input attribute, and these are updated in a consistent way, for example:

$$w(t+1) = w(t) + learning\_rate * (expected(t) - predicted(t)) * x(t)$$

The bias is updated in a similar way, except without an input as it is not associated with a specific input value:

$$bias(t+1) = bias(t) + learning\_rate * (expected(t) - predicted(t))$$

Now we can put all of this together. Below is a function named train_weights () that calculates weight values for a training dataset using stochastic gradient descent.

```
def train_weights(train, l_rate, n_epoch):
    weights = [0.0 for i in range(len(train[0]))]
    for epoch in range(n_epoch):
        sum_error = 0.0
        for row in train:
            prediction = predict(row, weights)
            error = row[-1] - prediction
            sum_error += error**2
            weights[0] = weights[0] + l_rate * error
            for i in range(len(row)-1):
                weights[i + 1] = weights[i + 1] + l_rate * error * row[i]
        print('>epoch=%d, lrate=%.3f, error=%.3f' % (epoch, l_rate, sum_error))
    return weights
```

You can see that we also keep track of the sum of the squared error (a positive value) each epoch so that we can print out a nice message each outer loop.

We can test this function on the same small contrived dataset from above.
You can see how the problem is learned very quickly by the algorithm. Now, let's apply this algorithm on a real dataset.

Task

1. Download the dataset of fruits from slate->lab#12 folder.
2. Remove the categorical/textual columns
3. Apply the built-in Perceptron algorithm from sciket-learn
4. Apply the Perceptron algorithm from scratch using above code snippets?
    - Apply on above dataset
    - Apply on fruit dataset