How To Implement The Perceptron Algorithm From Scratch In Python

by **Jason Brownlee** on November 2, 2016 in **Code Algorithms From Scratch**

Last Updated on August 13, 2019

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.

In this tutorial, you will discover how to implement the Perceptron algorithm from scratch with Python.

After completing this tutorial, you will know:

- How to train the network weights for the Perceptron.
- How to make predictions with the Perceptron.
- How to implement the Perceptron algorithm for a real-world classification problem.

Let's get started.

- **Update Jan/2017**: Changed the calculation of fold_size in cross_validation_split() to always be an integer. Fixes issues with Python 3.
- **Update Aug/2018**: Tested and updated to work with Python 3.6.

How To Implement The Perceptron Algorithm From Scratch In Python
Photo by Les Haines, some rights reserved.

Description

This section provides a brief introduction to the Perceptron algorithm and the Sonar dataset to which we will later apply it.

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
```

In this way, the Perceptron is a classification algorithm for problems with two classes (0 and 1) where a linear equation (like or hyperplane) can be used to separate the two classes.

It is closely related to linear regression and logistic regression that make predictions in a similar way (e.g. a weighted sum of inputs).

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 a function by following the gradients of the cost function.

This involves knowing the form of the cost as well as the derivative so that from a given point you know the gradient and can move in that direction, e.g. downhill towards the minimum value.

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.

Sonar Dataset

The dataset we will use in this tutorial is the Sonar dataset.

This is a dataset that describes sonar chirp returns bouncing off different services. The 60 input variables are the strength of the returns at different angles. It is a binary classification problem that requires a model to differentiate rocks from metal cylinders.

It is a well-understood dataset. All of the variables are continuous and generally in the range of 0 to 1. As such we will not have to normalize the input data, which is often a good practice with the Perceptron algorithm. The output variable is a string “M” for mine and “R” for rock, which will need to be converted to integers 1 and 0.

By predicting the class with the most observations in the dataset (M or mines) the Zero Rule Algorithm can achieve an accuracy of 53%.

You can learn more about this dataset at the UCI Machine Learning repository. You can download the dataset for free and place it in your working directory with the filename **sonar.all-data.csv**.

Tutorial

This tutorial is broken down into 3 parts:

1. Making Predictions.
2. Training Network Weights.
3. Modeling the Sonar Dataset.

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

1. Making Predictions

The first step is to develop a function that can make predictions.

This will be needed both in the evaluation of candidate weights values in stochastic gradient descent, and after the model is finalized and we wish to start making predictions on test data or new data.

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
```

We can contrive a small dataset to test our prediction function.

```
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.

```
# 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

# test predictions
dataset = [[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]]
```

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

There are two inputs values (**X1** and **X2**) and three weight values (**bias**, **w1** and **w2**). The activation equation we have modeled for this problem is:

```
activation = (w1 * X1) + (w2 * X2) + bias
```

Or, with the specific weight values we chose by hand as:

```
activation = (0.206 * X1) + (-0.234 * X2) + -0.1
```

Running this function we get predictions that match the expected output (**y**) values.

```
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=0, Predicted=0
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
Expected=1, Predicted=1
```

Now we are ready to implement stochastic gradient descent to optimize our weight values.

2. 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.

```python
# Estimate Perceptron weights 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.

```python
# 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

# Estimate Perceptron weights 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

# Calculate weights
dataset = [[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]]
l_rate = 0.1
n_epoch = 5
weights = train_weights(dataset, l_rate, n_epoch)
print(weights)
```

We use a learning rate of 0.1 and train the model for only 5 epochs, or 5 exposures of the weights to the entire training dataset.

Running the example prints a message each epoch with the sum squared error for that epoch and the final set of weights.

```
>epoch=0, lrate=0.100, error=2.000
>epoch=1, lrate=0.100, error=1.000
>epoch=2, lrate=0.100, error=0.000
>epoch=3, lrate=0.100, error=0.000
>epoch=4, lrate=0.100, error=0.000
[-0.1, 0.20653640140000007, -0.23418117710000003]
```

You can see how the problem is learned very quickly by the algorithm.

Now, let's apply this algorithm on a real dataset.

3. Modeling the Sonar Dataset

In this section, we will train a Perceptron model using stochastic gradient descent on the Sonar dataset.

The example assumes that a CSV copy of the dataset is in the current working directory with the file name **sonar.all-data.csv**.

The dataset is first loaded, the string values converted to numeric and the output column is converted from strings to the integer values of 0 to 1. This is achieved with helper functions **load_csv()**, **str_column_to_float()** and **str_column_to_int()** to load and prepare the dataset.

We will use k-fold cross validation to estimate the performance of the learned model on unseen data. This means that we will construct and evaluate k models and estimate the performance as the mean model error. Classification accuracy will be used to evaluate each model. These behaviors are provided in the **cross_validation_split()**, **accuracy_metric()** and **evaluate_algorithm()** helper functions.

We will use the **predict() and train_weights()** functions created above to train the model and a new **perceptron()** function to tie them together.

Below is the complete example.

```python
# Perceptron Algorithm on the Sonar Dataset
from random import seed
from random import randrange
from csv import reader

# Load a CSV file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
    class_values = [row[column] for row in dataset]
    unique = set(class_values)
    lookup = dict()
    for i, value in enumerate(unique):
        lookup[value] = i
    for row in dataset:
        row[column] = lookup[row[column]]
    return lookup

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
    for i in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        train_set.remove(fold)
        train_set = sum(train_set, [])
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            row_copy[-1] = None
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores

# 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

# Estimate Perceptron weights 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):
        for row in train:
            prediction = predict(row, weights)
            error = row[-1] - prediction
            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]
    return weights

# Perceptron Algorithm With Stochastic Gradient Descent
def perceptron(train, test, l_rate, n_epoch):
    predictions = list()
    weights = train_weights(train, l_rate, n_epoch)
    for row in test:
        prediction = predict(row, weights)
        predictions.append(prediction)
    return(predictions)

# Test the Perceptron algorithm on the sonar dataset
seed(1)
# load and prepare data
filename = 'sonar.all-data.csv'
dataset = load_csv(filename)
for i in range(len(dataset[0])-1):
    str_column_to_float(dataset, i)
# convert string class to integers
str_column_to_int(dataset, len(dataset[0])-1)
# evaluate algorithm
n_folds = 3
l_rate = 0.01
n_epoch = 500
scores = evaluate_algorithm(dataset, perceptron, n_folds, l_rate, n_epoch)
print('Scores: %s' % scores)
print('Mean Accuracy: %.3f%%' % (sum(scores)/float(len(scores))))
```

A k value of 3 was used for cross-validation, giving each fold 208/3 = 69.3 or just under 70 records to be evaluated upon each iteration. A learning rate of 0.1 and 500 training epochs were chosen with a little experimentation.

You can try your own configurations and see if you can beat my score.

Running this example prints the scores for each of the 3 cross-validation folds then prints the mean classification accuracy.

We can see that the accuracy is about 72%, higher than the baseline value of just over 50% if we only predicted the majority class using the Zero Rule Algorithm.

```
Scores: [76.81159420289855, 69.56521739130434, 72.46376811594203]
Mean Accuracy: 72.947%
```

Extensions

This section lists extensions to this tutorial that you may wish to consider exploring.

- **Tune The Example**. Tune the learning rate, number of epochs and even data preparation method to get an improved score on the dataset.
- **Batch Stochastic Gradient Descent**. Change the stochastic gradient descent algorithm to accumulate updates across each epoch and only update the weights in a batch at the end of the epoch.
- **Additional Regression Problems**. Apply the technique to other classification problems on the UCI machine learning repository.

Did you explore any of these extensions?
Let me know about it in the comments below.

Review

In this tutorial, you discovered how to implement the Perceptron algorithm using stochastic gradient descent from scratch with Python.

You learned.

- How to make predictions for a binary classification problem.
- How to optimize a set of weights using stochastic gradient descent.
- How to apply the technique to a real classification predictive modeling problem.

Do you have any questions?
Ask your question in the comments below and I will do my best to answer.

Tweet Share Share

About Jason Brownlee

Jason Brownlee, PhD is a machine learning specialist who teaches developers how to get results with modern machine learning methods via hands-on tutorials.
View all posts by Jason Brownlee →

< Python is the Growing Platform for Applied Machine Learning

How To Implement Learning Vector Quantization (LVQ) From Scratch With Python >