Modern Scientific Methods in Physics
Modelling Perceptrons using the Iris Data Set

Domonkos Haffner (S24Q2W)

March 19th 2021

Contents

1 Introduction

The research I chose for the semester was to create a simple perceptron model in $C++$ and find an appropriate data set to test it with. I chose the Iris data set, since it's perfectly adequate for the simplest perceptron model.

2 Iris data set

The Iris data set is is a multivariate data set, consisting of three different types of irises' petal and sepal lengths. The three species are the following:

1. Iris-setosa,
2. Iris-virginica,
3. Iris-versicolor.

The data set is a 150×5 matrix, containing the sepal lengths, sepal widths, petal lengths, petal widths and the names of the irises. [1] [2] I chose to write the perceptron model in order to distinguish the three types of species from each other.

Let's take a look at figure 1, where the different properties of the irises are represented as a scatter plot. As it's seen, the properties are very distinguishable for the Iris-Setosa - Iris-Versicolor and Iris-Setosa - Iris-Virginica pairs. This means that the simple perceptron algorithm should be quite effective. However, the Iris-Versicolor and Iris-Virginica properties are quite similar. This will result in quite similar predictions in the perceptron model. Note, that figure 1 was created in Python.

That data set can be easily found and downloaded from the internet. [3]

3 The model

After loading in the data set, I separated the values for the different properties and loaded it into the $C++$ code with a function.

Since this is a very basic Machine Learning project, the data set must be separated into training and testing data sets. I came up with an algorithm to separate the training and testing data according to a random normal distribution. At first this was completely random; I selected 20% of the data set and pushed it into the testing vector, then used the remaining 80% as the training data set. In order to get an equal amount of testing data for all groups, I took 0% of each iris group - with Mersenne Twister pseudo-random number generator - and used them as the testing data set. After this, I could easily select the remaining data set as training data.

All these lists of values are saved in $std :: vector < double >$ objects. Since the property vectors are consisting of double values, and the label vector is consisting of string values, the above mentioned train- and test data set generating functions must be written as template functions.

After splitting the data set, the perceptron model must be created. This is described by the following equation:

$$\text{y\_predicted} = Act(\underline{X} \cdot \underline{w} + b), \tag{1}$$

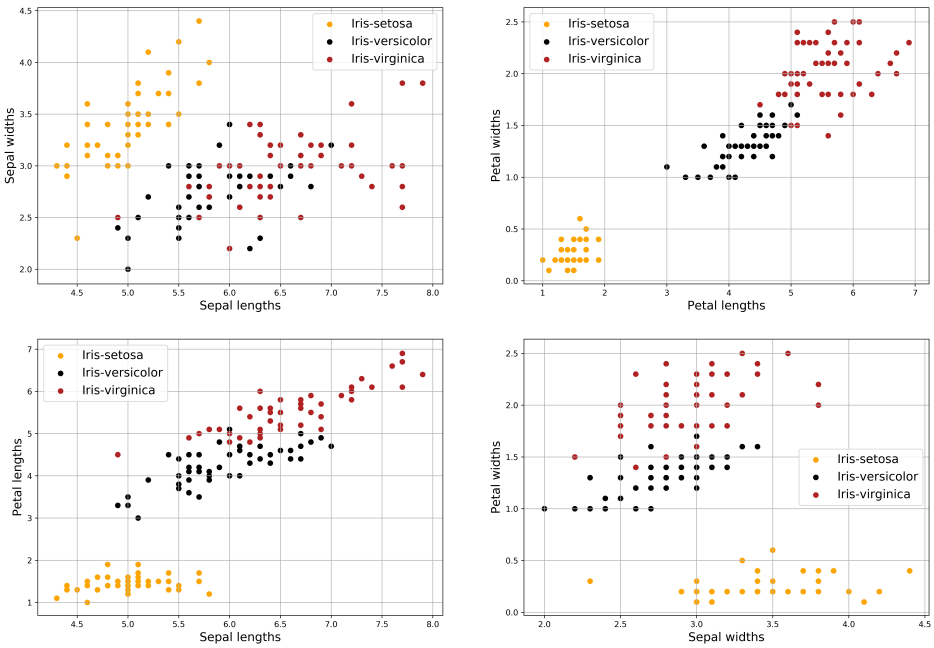

Figure 1: Scatter plot of the different properties of the irises.

where X is the property vector for each iris, w is the weight vector, b is the bias and Act is some kind of activation function. These weights and bias are initialised as zero. The activation function I'm using is the following:

$$Act(x) = 1, \text{ if } x > 0.5, \qquad Act(x) = 0, \text{ if } 0.5 \geq x > -0.5, \qquad Act(x) = -1, \text{ if } -0.5 \geq x, \quad (2)$$

where 1 represents the Iris-versicolors, 0 the Iris-virginicas and -1 the Iris-setosas. During every iteration, the y_predicted is calculated with this activation function. This is updated at every single iteration and used to update the weights in the following way:

$$\text{update} = \alpha \cdot (\text{y\_real[j]} - \text{y\_predicted[j]}), \quad (3)$$

$$\text{weights} \mathrel{+}= \text{update} * \text{x\_train[j]}. \quad (4)$$

where α is the learning rate, j is the row index of X and x_train a 4×1 vector containing the four properties of the irises.

After updating the weights, the classification function is called again, so a new prediction for the y_train data set is created. This iteration is repeated several times, which is set by the *epochs* variable. After the set epochs, the program finishes running and the weights are returned. These weights can be later used to predict the y_test classification, which can be compared with the actual testing labels.

4 The results

I used the following equation to calculate the accuracy values:

$$\text{accuracy} = 100 \cdot \frac{\text{correct\_pred}}{\text{correct\_pred} + \text{incorrect\_pred}}. \tag{5}$$

A function saved the accuracy values after every epoch, which were then wrote into a txt file. After loading these values in python, I created an accuracy plot, which can be seen in figure 2.

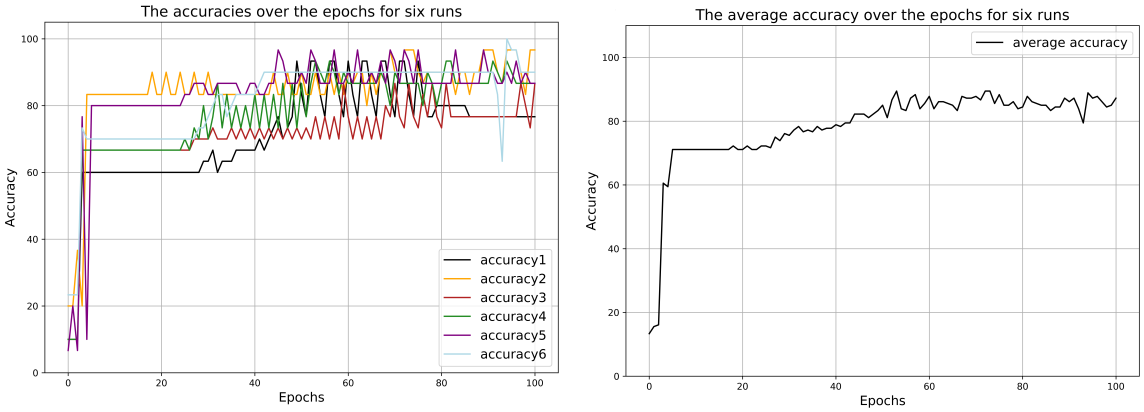

Figure 2: The accuracy of the model after 100 epochs (left) and the average of six runs (right).

As it's seen, using 100 epochs and $learning\_rate = 0.01$ I achieved about 90% accuracy with this perceptron model. I tried out different epochs and learning rates but these values seemed to be the most sufficient one. When using higher (about 0.1) learning rate, the results are oscillating much more, since the weights are changing by greater amounts.

5 Conclusion

In this research, a simple perceptron model was created to classify the different iris types into three groups; Iris-versicolor, Iris-virginica and Iris-setosa. This task was successfully done with mainly using $C++$ and using Python for plotting.

As assumed in the beginning of the research, the different iris types can be predicted from the four properties. The best results were given by a hundred epochs and $learning\_rate = 0.1$. The accuracy achieved by the model was about 90%.

References

[1] Wikipedia - Iris data set:
https://en.wikipedia.org/wiki/Iris_flower_data_set

[2] Scikit-learng documentation - Iris data set:
https://scikit-learn.org/stable/auto_examples/datasets/plot_iris_dataset.html

[3] Github - Downloading Iris data set:
https://gist.github.com/curran/a08a1080b88344b0c8a7