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

Domonkos Haffner (S24Q2W)

April 9th 2021

Contents

1 Introduction

The research I chose for the semester was to create a multiclass 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 this 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. The indices of the data were generated 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:

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

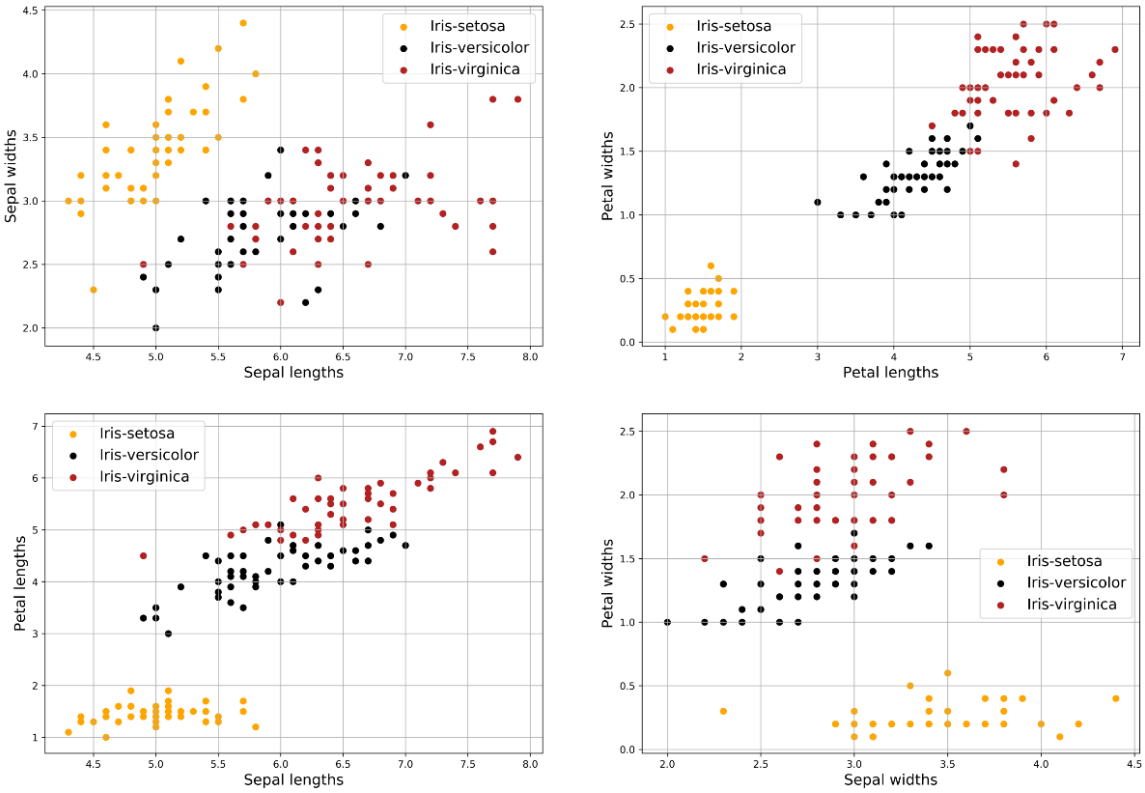

Figure 1: Scatter plot of the different properties of the irises.

where X is the property vector for each iris, w is the weight vector, b is the bias. Since X contains the features for all the 150 samples, it's a 150×4 matrix. The *weights* vector must be chosen in correspondence to the amount of features and classes the data set has. This results in a 5×3 *weights* matrix, where 5 comes from the four features (weights) and one bias, while the 3 means the number of classes. The *weight* matrix is initialised with a pseudo-random number generator taking values between 0 and 1.

After the perceptron model, an *Argmax* activation function is used. This takes the 150×3 resulted matrix and chooses the index of each column, which contains the highest value. This results in a 150 long vector, which contains the predicted values.

The next step is to update the weights in correspondence to the resulted predicted values. The methodology for this is seen in the following equations:

$$\begin{aligned} w_{i,ypred}(t+1) &= w_{i,ypred}(t) + \alpha x_i, \\ w_{i,yreal}(t+1) &= w_{i,yreal}(t) - \alpha x_i, \end{aligned} \tag{2}$$

where $i \in \{0, 150\}$ is the number of samples, $t \in \{0, 150 \times epochs\}$ is the time steps the weights are updated, α is the learning rate, $ypred$ is the predicted y value, $yreal$ is the real y value and x_i is the current sample. Note, that when $ypred = yreal$, the same value increases and decreases, so the weight remains unchanged. [4] [5]

The cost value is simply calculated by the following value:

$$\sum_i \Big(Max(Argmax(\underline{X} \cdot \underline{w} + b))_i - (Argmax(\underline{X} \cdot \underline{w} + b))_{i,yreal} \Big) + \Big(\sum_{c,n} abs(weights_{c,n}^2 \Big)^{1/2}, \tag{3}$$

where c is the number of classes and n is the number of features. [6]

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:

$$\text{accuracy} = 100 \cdot \frac{\text{correct\_pred}}{\text{correct\_pred} + \text{incorrect\_pred}}. \tag{4}$$

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- and cost function plot, which can be seen in figure 2.

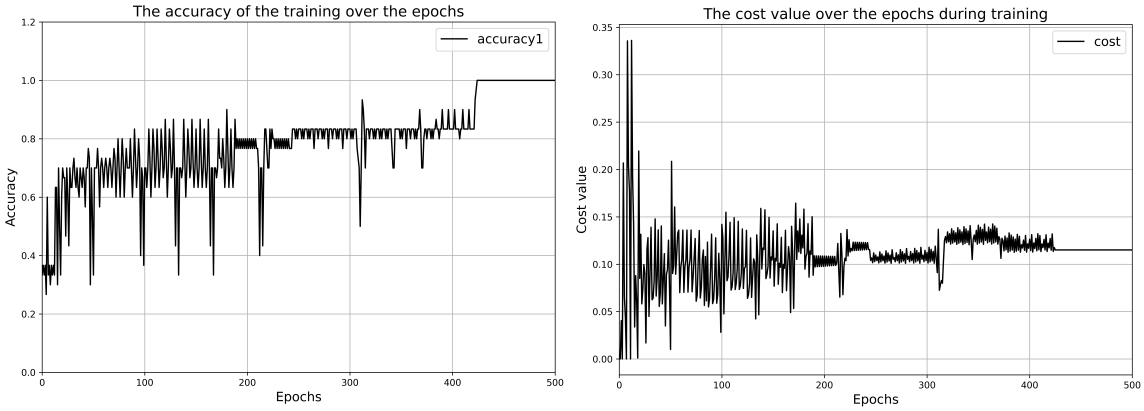

Figure 2: The accuracy of the model when predicting the results (left) and the cost function during training (right) over 500 epochs.

When looking at figure 2, one can tell that after about 420 epochs, all the labels were predicted correctly for the training data set, so neither the weights, nor the loss fluctuates anymore. This means that the optimal weights were set for predicting the training data set. Using this for the testing data set will not give 100% accuracy, however the results should be sufficient. Let's take a look at the accuracy plot of the training data set in figure 3.

As it's seen, using 500 epochs and $learning\_rate = 0.001$ I achieved about 97% accuracy with this multiclass perceptron model. I tried out different epochs and learning rates, which can be seen in the conclusion chapter. Note, that after about 400 epochs, the results are already adequate, however, in other cases more or less epochs were needed. Another plot can be created, which shows a couple different runs and the average of the accuracy values. This can be seen in figure 4.

As it's noticeable in figure 4, the accuracy function is much smoother than taking only one run into consideration. I created another plot, which shows the average of 100 runs over 500 epochs. This can be seen in figure 5.

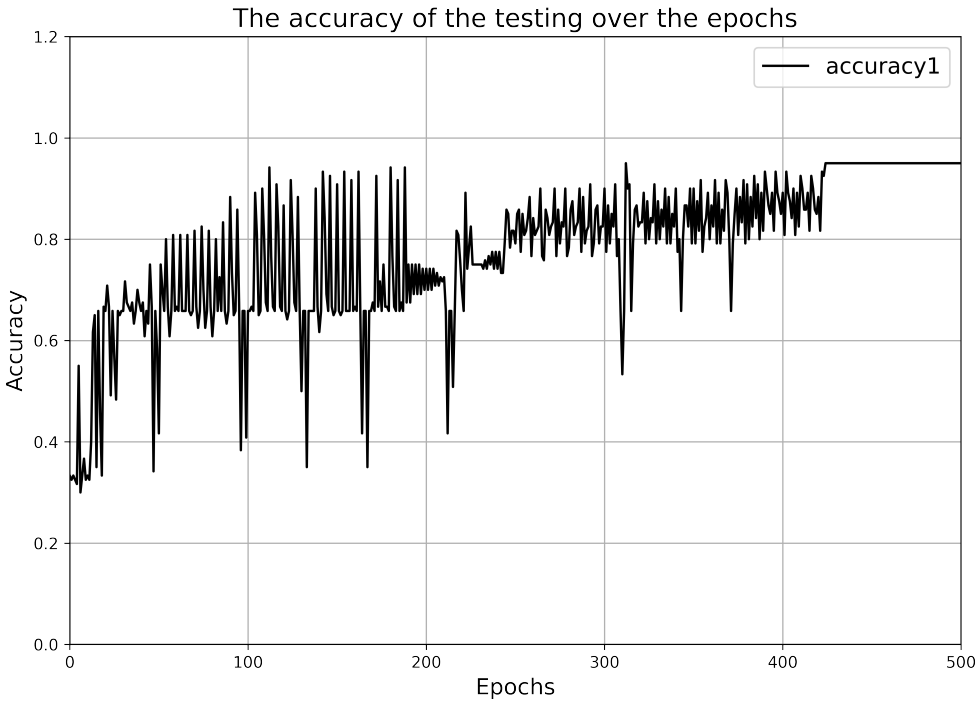

Figure 3: The accuracy of the model when predicting the results for the testing data set.

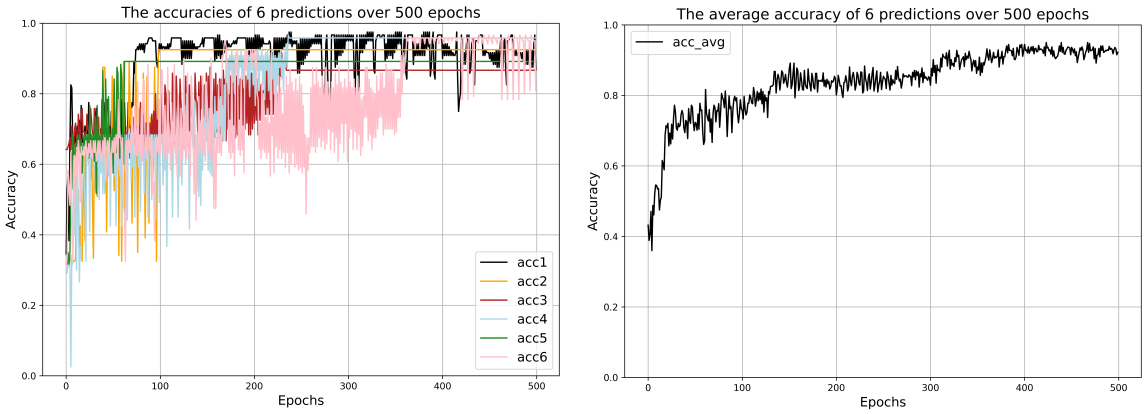

Figure 4: The 6 different- (left) and the average accuracy (right) of the model for the testing data set over 500 epochs.

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 Python for plotting.

As assumed in the beginning of the research, that the different iris types can be predicted from the four properties. The results with different learning rates and epochs can be seen in table 1.

Note, that table 1 is containing the average results of 100 runs for each epoch and learning rate.

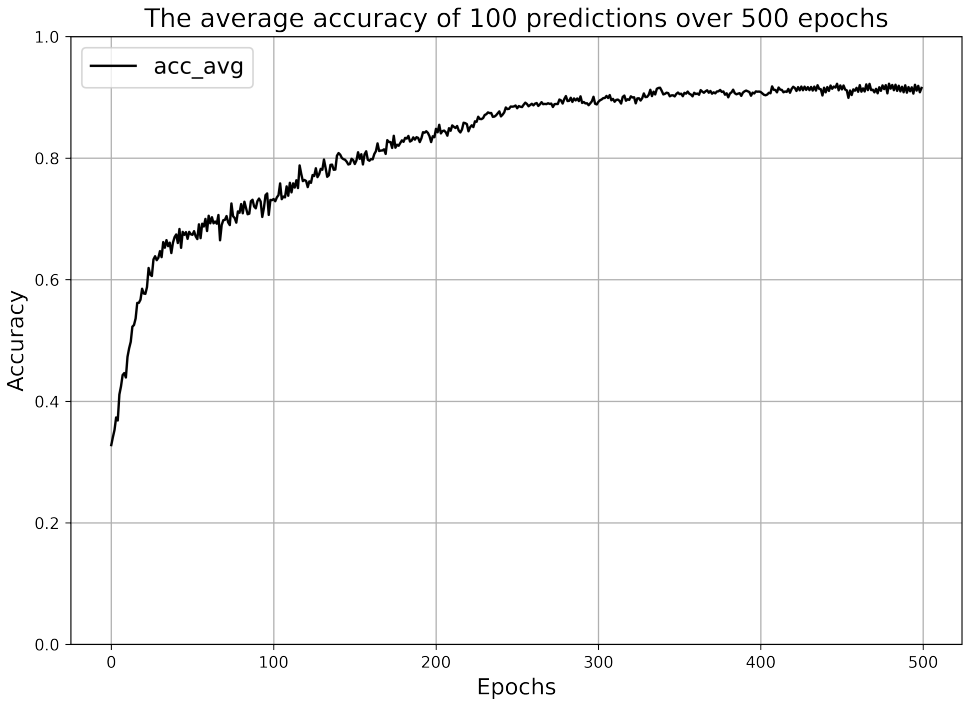

Figure 5: The average accuracy of 100 runs for the testing data set over 500 epochs.

As it's seen, it's not worth running the code for only a 100 epochs, since the results are only going to be around 67% accurate at most. Running the model till 500 epoch might be the ideal choice with a quite high learning rate, since it doesn't require a lot of time and give about 95% accuracy with a learning rate about 10^{-1} or 10^{-2}. Running the code for a 1000 epochs can also result in quite good predictions, however one must note that it takes more time to run the code.

There could be further steps in investigating the multi class perceptron; one could search for a data set, which contains many more samples and classes, however, this research only focused on the iris data set, and thus, is concluded.

	Epochs = 100		Epochs = 500		Epochs = 1000	
Elapsed time [ms]	645		1721		4122	
	Train	Test	Train	Test	Train	Test
Learning rate = 10^{-1}	0.9440	0.6548	1	0.9632	1	0.8631
Learning rate = 10^{-2}	0.9413	0.6501	1	0.9621	1	0.8781
Learning rate = 10^{-3}	0.7693	0.6655	0.9860	0.9426	1	0.9335
Learning rate = 10^{-4}	0.5933	0.5399	0.8520	0.7751	0.9173	0.8435

Table 1: Average accuracy values over different epochs and learning rates for the testing data set.

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

[4] Github - Multi-Class Classification and the Perceptron:
https://jermwatt.github.io/machine_learning_refined/notes/7_Linear_multiclass_classification/7_3_Perceptron.html#Regularization-and-the-multi-class-Perceptron

[5] Youtube - Relating the Binary and Multi-class Perceptron:
https://www.youtube.com/watch?v=HpPBDGYkYXQ&ab_channel=BertHuang

[6] Numpy Documentation - Linalg Norm:
https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html