NEURAL NETWORKS AND GENETIC ALGORITHMS

BALANCE SCALE DATA SET

CLASSIFICATION

BRATUCU ANA MARIA

GROUP 1231EA

**Multi-layer Perceptron Method**

The chosen Balance Scale data set reveals information about all the possibilities of positions a balance can have, taking into consideration the weights and their distance on each pan. Given this information, we have to predict if the scale is tipped or if it is balanced. The features represented by the weighs and distances are values from 1 to 5 and the class attribute consists of a column of strings. The scale is in balanced position if the multiplication of the weight and distance on each pan are equal.

Data set link: http://archive.ics.uci.edu/ml/datasets/Balance+Scale

Attribute details:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| left-weight | numerical | weight on left pan (1, 2, 3, 4, 5) |
| left-distance | numerical | distance on left pan (1, 2, 3, 4, 5) |
| right-weight | numerical | weight on right pan (1, 2, 3, 4, 5) |
| right-distance | numerical | distance on right pan (1, 2, 3, 4, 5) |
| balance | string | which way the scale tips (l – left, r – right,  b – balanced) |

There are a total of 625 instances in the data set, meaning that it covers all combinations in which weights from 1 to 5 can be placed onto the pans at certain distances (1 – 5). There are 49 cases in which the scale is balanced and 288 for each case in which it is tipped (to the right or to the left). Therefore, the data set is complete, there aren’t any new instances that can be added.

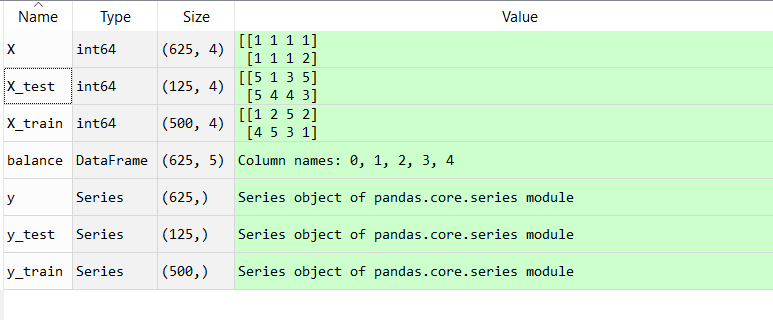
Given the type of the data set, this is a problem that must be associated to the classification type of Machine Learning, because we have classes (‘B’, ‘L’, ‘R’) and the based on the inputs of weights and distances, we classify data into one of them by programming the computer to learn the general rule (supervised learning).

Therefore, we build a neural network. Luckily for us, this set doesn’t contain missing values, so we don’t have to clean the data. For this classification problem’s algorithm, I use:

* the Scikit-learn implementation of the Multi-layer Perceptron classifier (MLPClassifier) to generate my the Perceptron algorithm
* the pandas library for handling the data set (read the .csv file from my computer)
* the matplotlib library to plot the error of the neural network
* the numpy library for the mathematical calculations

I start off by breaking the balance scale data set into one with features and another one only with the column of the class.

In order to avoid overfitting, I split the features data set into two subsets: for the training and for the test (test\_size=0.2 => 20% from the whole set for testing and the rest of 80% for training). This way, when I make the test, I make sure that the network won’t return the right output just because it trained too well and memorized the data it was trained on. If that happens, the performance will decrease on completely new data point.



Afterwards, I create the MLPClassification function, build the Multi-layer Perceptron classifier from the training set (X, y) and train the model:

mlp=MLPClassifier ( activation='relu',verbose=True,hidden\_layer\_sizes=(5,5,5),

max\_iter=5000)

mlp.fit(X\_train,y\_train)

* activation='relu' is by default
* verbose=True is for printing progress messages
* I chose hidden\_layer\_sizes to be (5,5,5) because I want to have 3 layers with 5 neurons each as I have 5 columns in the data set.
* max\_iter=5000 represents the number of steps; initially I put 500, but because that wasn’t enough, I increased it to 5000, although the iterations never passed a maximum of 1500

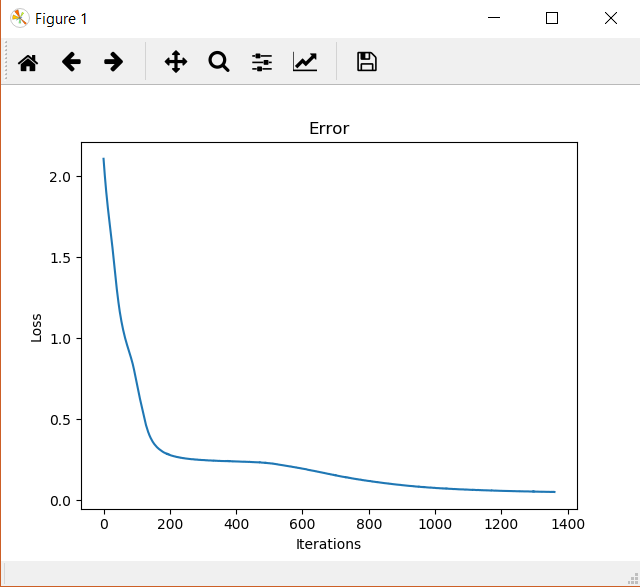
Now that the neural network prepared itself with the training set and learned its patterns, it’s time to predict the outcome for the testing set:

predictions = mlp.predict(X\_test)

and compute the error:

err=mlp.loss\_curve\_ , where loss\_curve\_ is a built-in function for the MLPClassifier for finding the error of the Perceptron.

Then, I plot it



This is a good sign as we can observe, the plotted error is high when the iterations first start and decreases inverse proportionally as iterations increase, finally getting very close to the 0 value at iteration 1361 and remaining unchanged from that point on. This means that the algorithm was able to successfully predict most of the outcome for the testing set.

To clearly see the accuracy for our program, we compute it either by using the formula:

acc=100 \* np.sum((y\_test == predictions )/len( y\_test ))

or the function:

acc1=accuracy\_score(y\_test, predictions)\*100

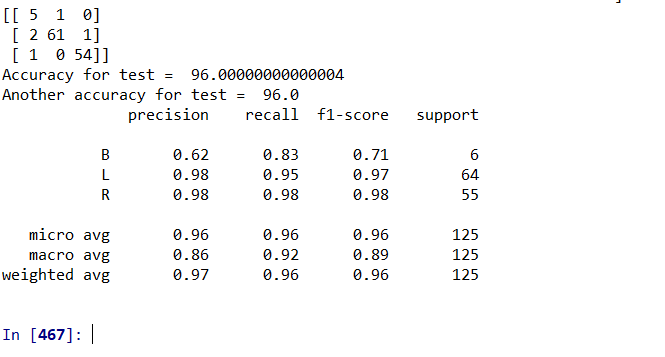
Also, using sklearn.metrics, I generated the classification\_report and the confusion\_matrix

to further evaluate how well our model performed:

print(confusion\_matrix(y\_test,predictions))

print(classification\_report(y\_test,predictions))

This is the output:



As we can see, the accuracy, determined with both methods, is 96% which is almost perfect.

In the matrix, the values from the diagonal are the correctly predicted outputs for each class (B, L, R). From it, we get the information that out of 6 cases of ‘Balanced’ scale, 1 has been misclassified, out of 64 cases of ‘Left’ scale, 3 has been misclassified and out of 55 cases of ‘Right’, 1 has been misclassified. In total, we have only 5 incorrect results from 125 data points on which we tested our neural network.

To further verify that I have good results, I manually give an input to train to the network and the output I know I have to obtain:

t=['B']

x=[[1,1,1,1]]

print('For taget ', t, 'we get ', mlp.predict(x))



Success!

In order to get the best results, we want to normalize the data, so there is a standard format for it. That way, the training is improved, more accurate and faster. Using sklearn’s method, StandardScaler, I scale the data (training and testing sets):

scaler = StandardScaler()

scaler.fit(X\_train)

X\_train = scaler.transform(X\_train)

X\_test = scaler.transform(X\_test)

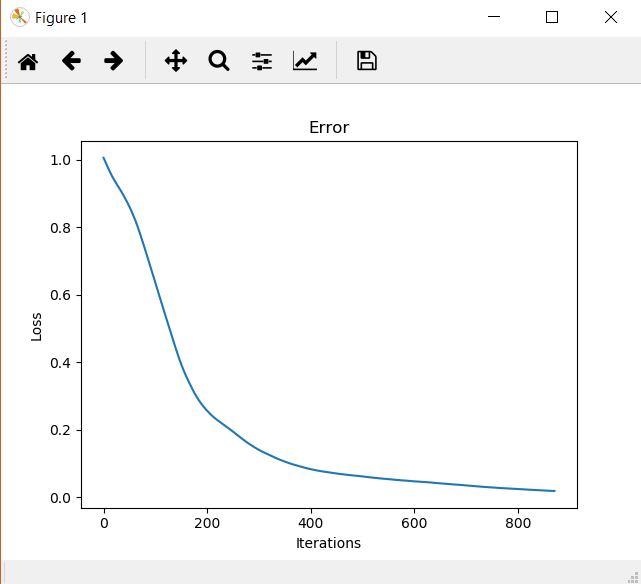


Fig. 1

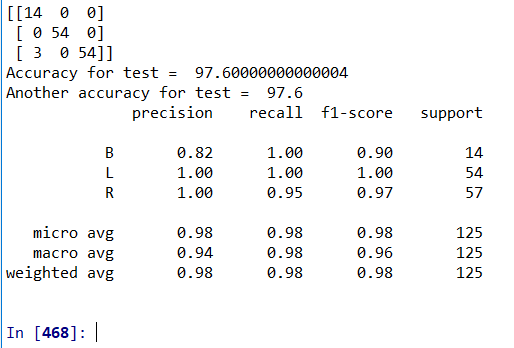


Fig. 2

From the plot (Fig. 1) we observe that the error is lower (0.019) obtained in a faster time (only 872 iterations) than the first run, before scaling the data. Therefore, we see in Fig. 2 that the classifier was right 97.6% of the time, which is an increase from the previous result.

Let’s change the parameters of the MLPClassifier function to see if we can make it even more accurate.

mlp =MLPClassifier( activation='relu',

verbose=1,hidden\_layer\_sizes=(5,5,5),

max\_iter=5000,**learning\_rate='adaptive'**,

**solver='sgd'**,**learning\_rate\_init = 5**)

I added:

* **learning\_rate='adaptive'** which is constant to initial learning rate (**learning\_rate\_init = 5**) as long as training loss keeps decreasing; this is used with **solver='sgd'**

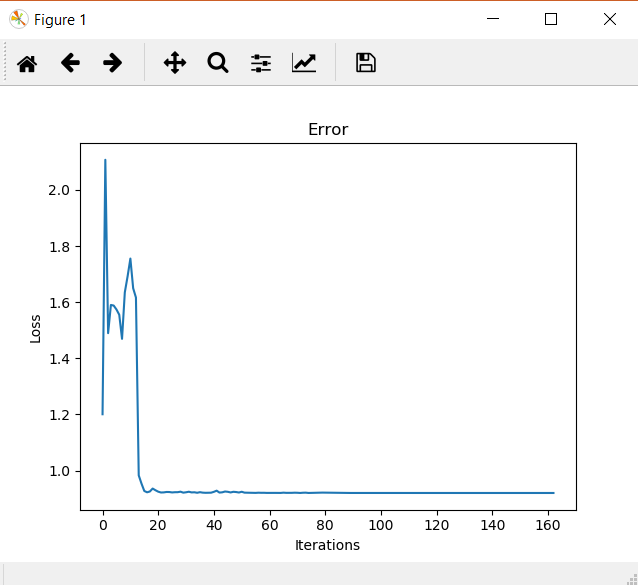


Fig. 3

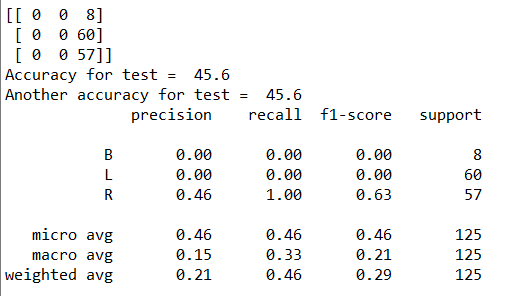


Fig. 4

Using these settings we observe in Fig. 3 that the number of iterations are considerably lower (161) than the first two tests, but the accuracy also decreased drastically to 45.6%. We see in Fig. 4 that all the cases when the balance scale is either in Balance or tilted to le Left are misclassified. That means that the neural network didn’t pass through enough data points with class of ‘B’ and ‘L’ when it has made its training. Therefore, couldn’t extract enough information and arrive at a conclusion regarding those outputs, so they were all mistaken in the test data set.

Another important parameter is the hidden\_layer\_sizes, so I changed the initial tuple that I had (5,5,5) to higher values:

mlp = MLPClassifier(verbose=1,hidden\_layer\_sizes=(13,13,13),max\_iter=5000)

These are the results:

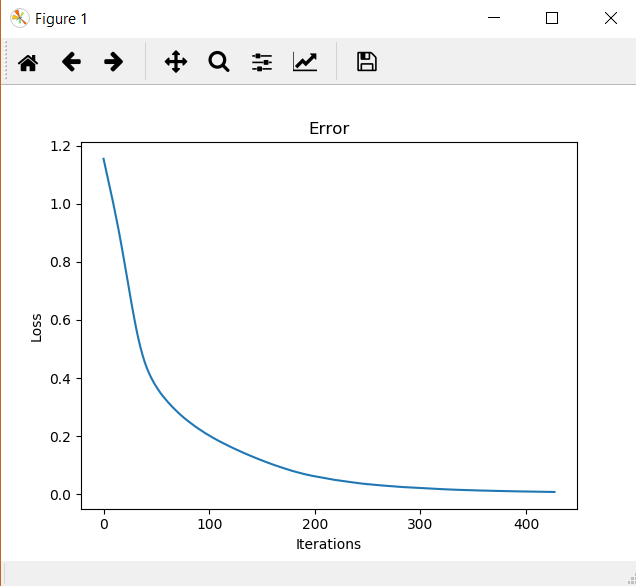


Fig. 5

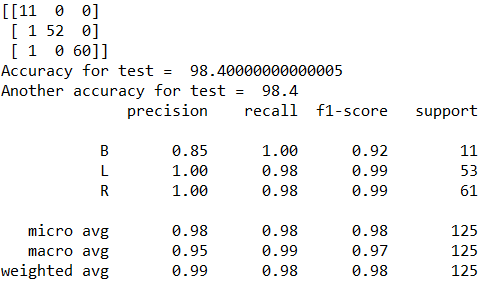


Fig. 6

This is the best outcome that I’ve had so far. Only increasing the layers per architecture not only decreased the loss to a value that we can approximate to 0, but did it fast, in 428 iterations. And the accuracy is 98.4% with only 2 misclassifications, making this the most accurate neural network that I could obtain for this algorithm and data set.

**DECISION TREE METHOD**

Another instrument to train a neural network is the Decision Tree. Using supervised learning, it predicts the value of a target by creating a tree like model and using it for learning decision rules indicated from the data features. This is how it is different from the MLPClassifier method presented previously.

For the implementation part, I will use the same libraries and methods as before. I upload the data set and split it into train and testing data, then adding the DecisionTreeClassifier function that will compute the algorithm needed for this method. Moving forward, I build a decision tree classifier from the training set and based on the patterns extracted, the neural network will predict the results for the testing set:

tree = DecisionTreeClassifier(min\_samples\_split=3, splitter='random',

criterion='gini')

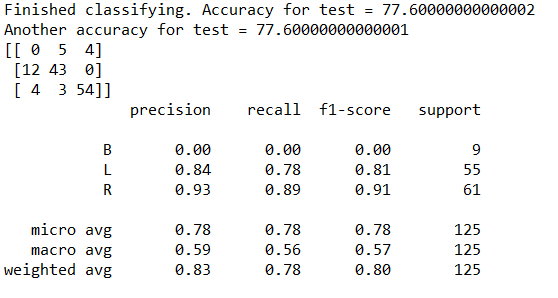
tree.fit(X\_train,y\_train)

predictions = tree.predict(X\_test)

I use the parameters:

* min\_samples\_split=3 for minimum number of samples required to split a node; I chose it to be 3 because I didn’t want the tree to be too constrained as if it was a higher number, it would’ve had to consider more samples at each node
* splitter='random' because every feature in the data set is relevant to the output, so there is no to use the ‘best’ split because there isn’t one
* criterion='gini' is by default, but I introduced it to highlight the fact that I want to measure the impurity of the nodes (when all of its records belong to the same class = leaf node)

Afterwards, I compute the accuracy as before and print the confusion matrix and classification report



I obtained an average accuracy of 77.6% with 0 good classifications when the balance scale found itself in a balanced position.

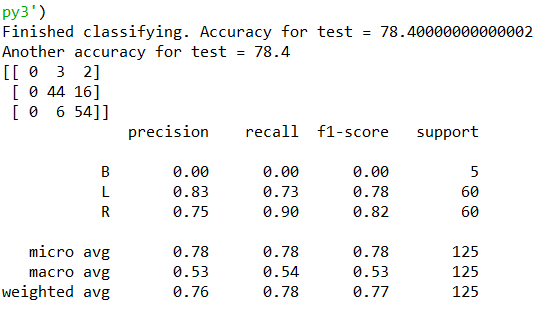
In order to obtain a better accuracy, I added more parameters to the DecisionTreeClassifier function to make it more customized, to find a balance between making the tree less complex and making it complex enough to not lose sight of the smaller patterns in the data:

tree = DecisionTreeClassifier(min\_samples\_leaf=30,max\_leaf\_nodes=70,

min\_samples\_split=3,splitter='random',criterion='gini'),

where

* min\_samples\_leaf=30 - the minimum sample size can be fixed to 30 so that’s what I chose because I wanted to restrict the size of sample leaf
* max\_leaf\_nodes=70 - because my tree is complicated enough and I wanted to simplify it by reducing the number of leaf nodes, expecting e better accuracy



And my assumptions were correct, as there can be observed a slightly increased change for the accuracy value (78.4%). But the precision for the Balanced class is still 0, with every case wrongly classified for it. That is a sign that the model doesn’t take into consideration enough information to accurately model completely new data.

In order to correct it, I pruned the tree by setting a value for max\_depth parameter of DecisionTreeClassifier function. However, I wanted to see the evolution of the accuracy for every value for max\_depth from 2 to 14 by computing it at every step and then plotting the graph of it. For better observations, I also plotted the graph of accuracies for the training data, to see what the result should’ve been.

tree = DecisionTreeClassifier(**max\_depth=i**,random\_state=0,

min\_samples\_split=10,splitter='random')

* **max\_depth=i** will take every value from the range (2,15) and create a tree with different depths

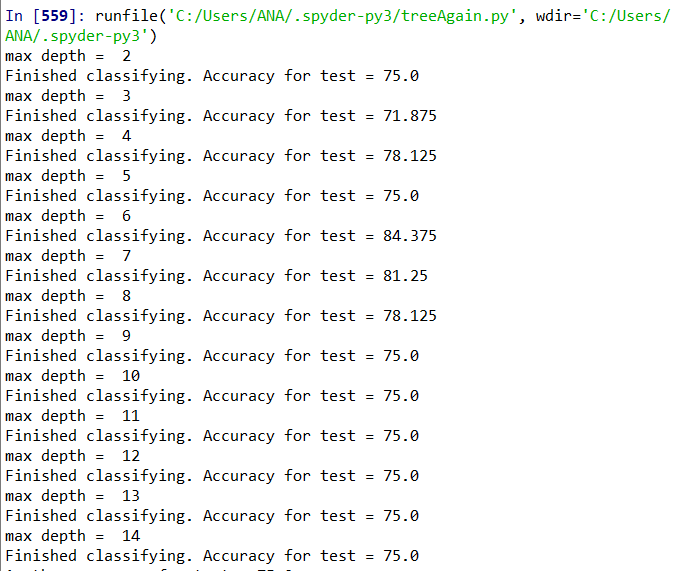


Fig. 1

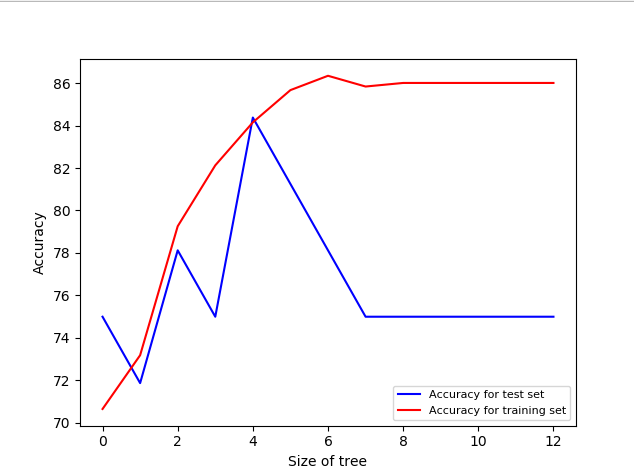


Fig. 2

From Fig. 1 it can be seen that the accuracy varies from 75% to a high of 84% reached max\_depth = 6, that making it the most efficient value for this parameter and for the neural network in general because this setting shows me the highest accuracy I can arrive at with Decision Trees.

Fig. 2 is a clear sign that our model overfits for the depth values chosen. Especially from size of tree = 7 up to 14 where there is a big difference from the training test. The tree predicts well all the training data, however it fails to generalize for new data, hence it commits errors. The solution to stop the problem is to choose the max\_depth to be 6 so there won’t exist overfitting or underfitting.

In conclusion, after using MLPClassifier and DecisionTreeClassifier methods to train the neural network to find in what state the balance scale finds itself in (balanced, tilted to the left or to the right), it is clear to say that the Perceptron did a very good job for this data set in particular. That is because we managed to predict the correct class for each data point in proportion of 98.4% compared to the maximum accuracy of 84.3% for the Decision Tree. But it is a known fact that Trees are prone to overfitting and they can be unstable. On the upside, the data didn’t need any preparation in advance, compared to the MLPClassifier where I had to scale the data.

**Bibliography**

1. https://scikit-learn.org/stable/modules/generated/sklearn.neural\_network.MLPClassifier.html
2. https://scikit-learn.org/stable/modules/tree.html
3. https://towardsdatascience.com/deep-learning-overfitting-846bf5b35e24
4. https://subscription.packtpub.com/book/big\_data\_and\_business\_intelligence/9781788397872/5/ch05lvl1sec62/scaling-of-data-in-neural-network-models