T.E.I. of Crete

Artificial Neural Networks

(laboratory course)

Final project

Yenokyan Lilit

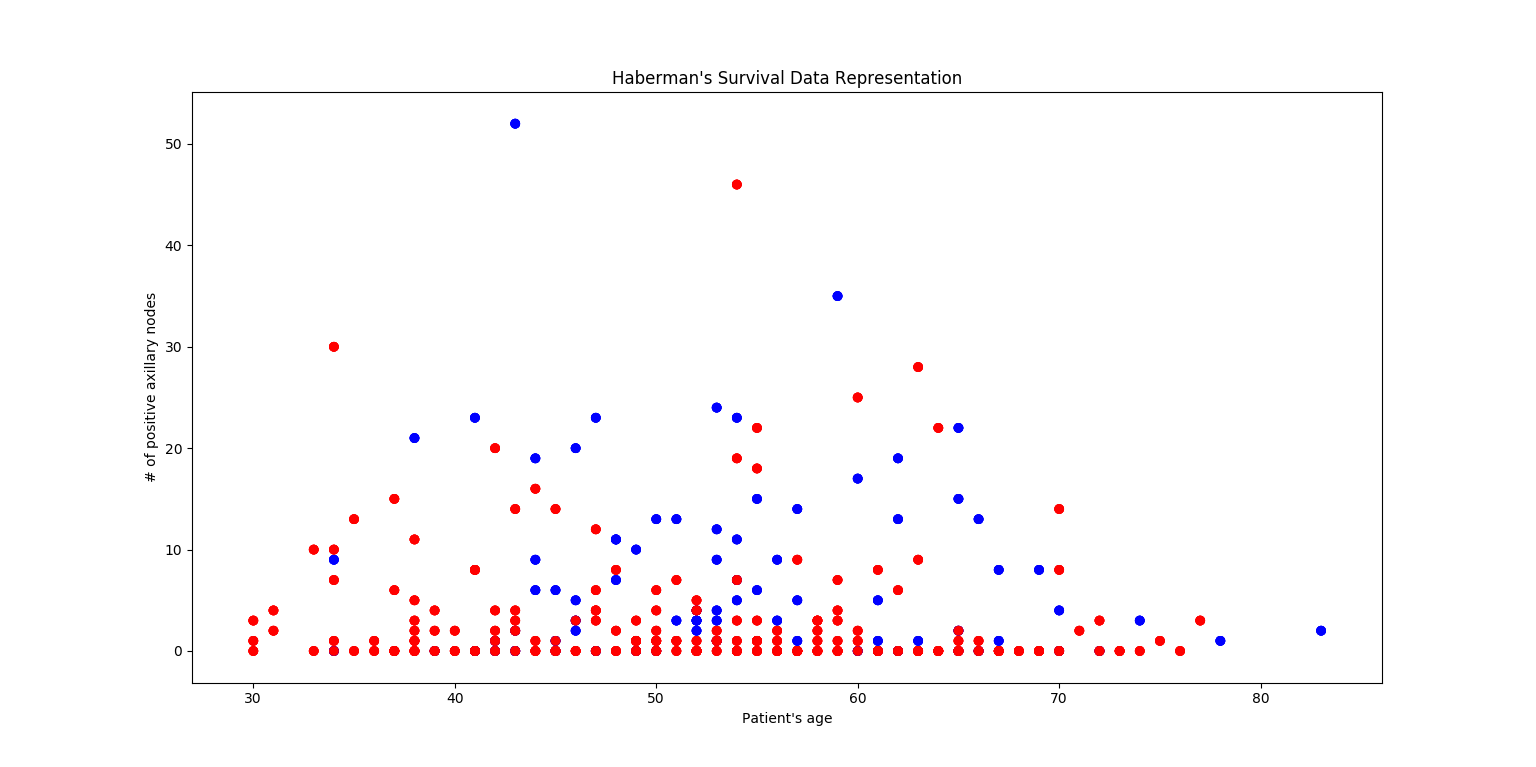
5/31/2019

This paper will explain and illustrate the results obtained by creating and training an artificial neural network using the **Haberman’s Survival dataset** ( <https://archive.ics.uci.edu/ml/datasets/Haberman%27s+Survival> ). The data gives information of the patients who had undergone surgery for breast cancer. It contains patients age, year of operation and the number of detected positive axillary nodes as well as the survival status of these people.

I believe that it is worth mentioning that this dataset has no missing values. Otherwise it would affect the results of the prediction of our ANN negatively. Also, all the given data is in numerical form meaning calculations can be done by using simple functions and no need for transforming categorical values. Only the survival status is a class attribute (1 or 2).

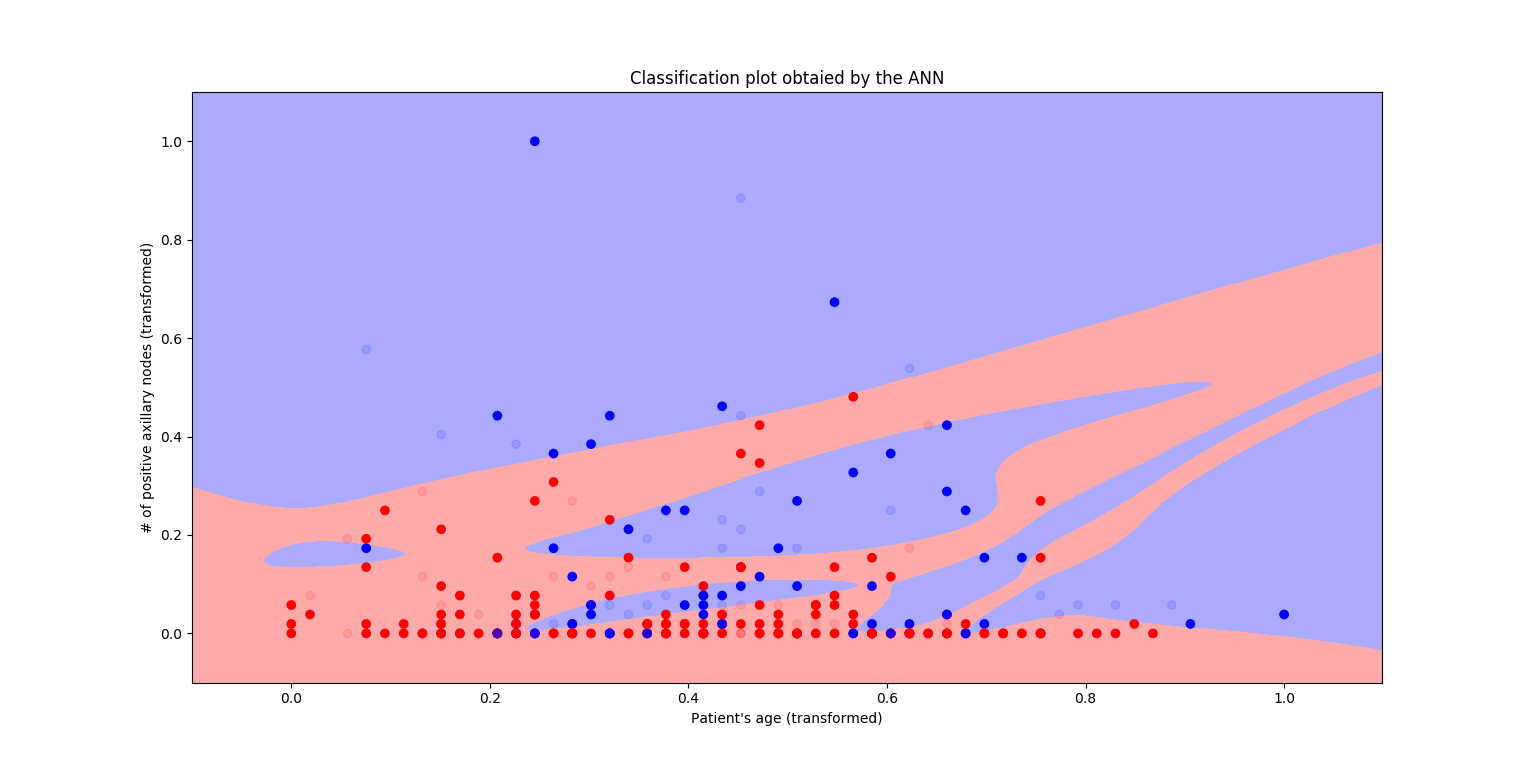
The programming language used for obtaining, training and testing the ANN model is **PYTHON 3.5.2.**

The neural network is created using Python’s sklearn module. All the necessary modules are imported for processing the given dataset, creating the neural net., testing it and plotting/analyzing the results.

First, I am importing the data and giving the descriptive names to each column respectively. Then separating features (age, year and number of nodes) from the class attribute (survival result). As required the data is displayed using *matplotlib.pyplot.* The result is displayed below: 

The red dots of the graph represent the patients who survived and the blue ones those who did not. The X coordinate of each dot is the actual age of the patient and the Y is the number of positive axillaries nodes he had. In the similar fashion one can obtain the plot depending on any other two features. After visualizing and understanding the data we are given, it is time to create the model. First step is to preprocess the data and we start by scaling it. By using the *MinMaxScaler* function we bring the values of all features to the range [0,1] and transform it.

Afterwards we need to split data at least into two parts one of which will be used for training the model, another one for testing it. As required, we start by splitting features set into training and testing sets with 70 and 30 percent ratios respectively. After having the datasets ready it is time to build the model itself. The requirements of the model are the following: the number of epochs is 1000, the learning rate is 0.3 and the performance goal is 1e-5. The function used for creating the model is *sklearn’s MLPClassifier.* The number of layers used is 2 and each layer has 15 neurons. These numbers were chosen randomly based on the intuition and experience. All the parameters mentioned earlier are set according to the function requirements. Additional parameter *early\_stopping* is set to True which will automatically set aside 10% of training data as validation and stop the training if the validation score is not improving by the tolerance. After the model is ready, we fit the training data to train the model. All the next steps are for plotting the final results: create the color map, create the mesh (those ±0.1s are added just to make the frame a bit bigger), predicte Z values on mesh grid to generate decision boundaries, plot the training data and plot the testing data in lighter color on the same plot. After preparing the mesh we create the plot, give respective labels for axis and a title. The final result of the initial run is illustrated below:

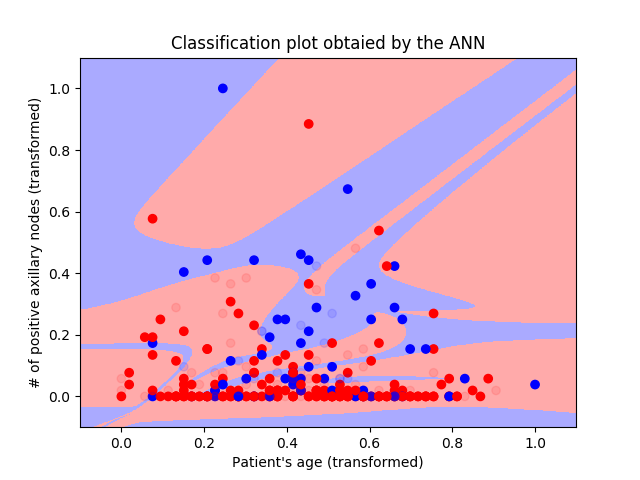
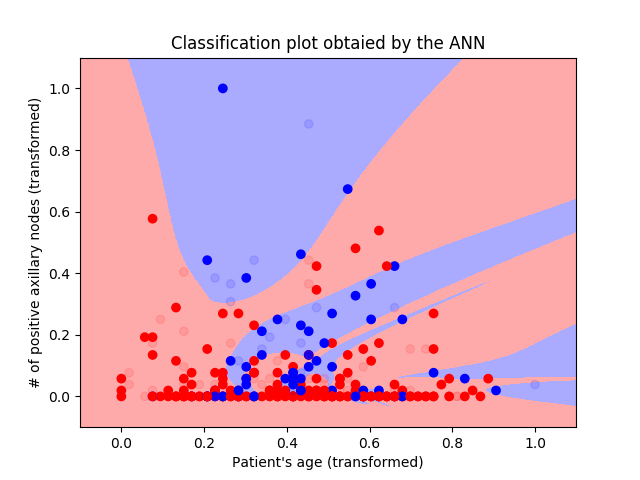


As mentioned on the plot the values on the axis are transformed and not the actual ones. The light-colored dots are testing results and the bold ones are trainings.

At the end we get the accuracy score of our model which for this specific case was 71.74% or (0.717381430 the actual). I believe this is a fairly good result keeping in mind the fact that it is just the first trial. Besides the score we can also get information about the true or false guesses of the model (confusion matrix) using the *confusion\_matrix* function. This is a table used for describing the performance of the classification models. From the matrix we get the information about false positives, false negatives, true positives, and true negatives. In this specific case the matrix is the following: [[53 14] [12 13]] meaning we have 53 + 13 correct and 14 + 12 wrong predictions.

As the model is created, trained and tested it is time to experiment.

Next try is to increase number of layers from 2 to 3 (reminder: each layer has 15 neurons). The results were much better. I observed 76% accuracy and the plot of classification is attached below (left one):



Next, I tried to decrease the number of neurons in all layers: BAD IDEA, the accuracy score dropped to 62% and the confusion matrix is [[52 12][22 6]]. It is interesting to notice that the number of positive values (both true and false) is not changing much, it is mostly the negative values that change. And this change in the negative values is affecting the accuracy score. Anyway, the messy plot of the described change can be seen above on the right.

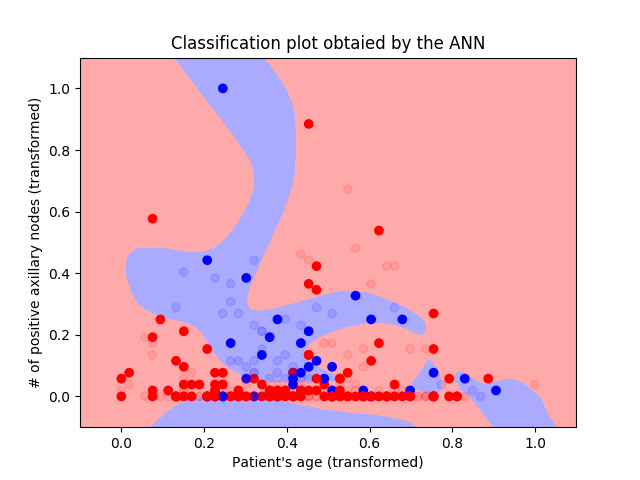
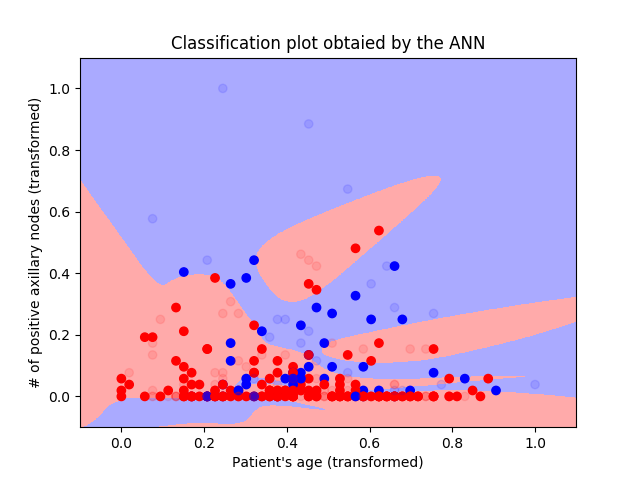
I even changed the number of layers from 3 to 1. The results were somehow satisfying. The model almost never gave predictions with higher score than 71% but also no score lower than 65% which indicated that the model’s predictions are quite stable and this is a positive sign.

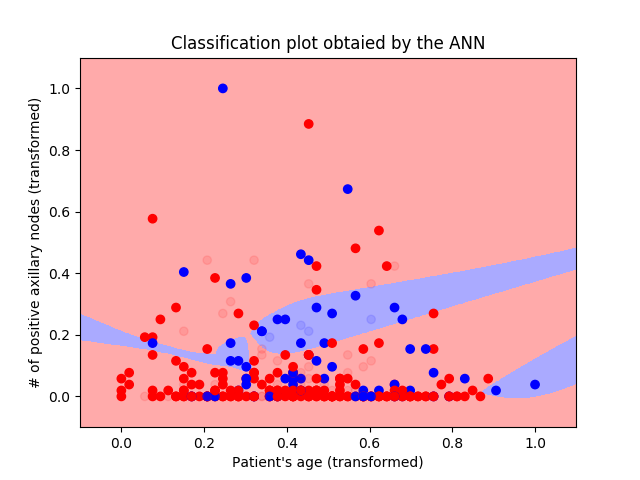
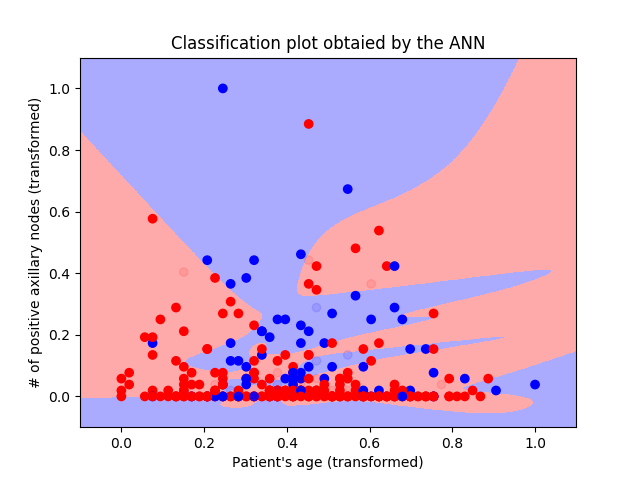
However, I believe I obtained the best results by using 2 layers. And after many changes and trials in the numbers of neurons in each layer I observed more or less stable and satisfying results with the model that has 2 layers with 20 and 10 neurons respectively. It is 77% accurate and has [[66 8] [13 5]] confusion matrix.

As required the table below shows the results, I obtained from different ratios of data split using 2 layers with 20 and 10 neuros:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test Number** | **Split Ratios** | **Accuracy Scores** | **Confusion Matrices** |
| **1** | 70%-30% | *77.17%* | [[66 8]  [13 5]] |
| **2** | 50%-50% | *68.63%* | [[89 20]  [28 16]] |
| **3** | 60%-40% | *73.17%* | [[77 11]  [22 13]] |
| **4** | 80%-20% | *70.96%* | [[37 8]  [10 7]] |
| **5** | 90%-10% | *51.61%* | [[14 3]  [12 2]] |

**The plots of the experiments described in the table above are below:**

** 2. 3.**

**4. 5.**

Overall, it is clear from the table that the best results are obtained when the data set is devided with the 70-30 ratio and the worst when we use only 10% of the data for testng. This is very obvious because althogh the model is well trained(90%), the number of testing instances is so small that every mistake playes very big role on the overall picture of the prediction.

To sum up, I believe that there is no universally true and only one way to train neural networks in the matter of layer and neuron numbers, and feature usage. Everything depends on time, training and experimenting. There are many factors that must be taken into account like number of input and output neurons, whether it is supervised or unsupervised leanrning, the activation functions, etc. But again, more I experimented the better my understanding of the spesific case was and I was obtaining much better results.