NN-based regression for predicting the clinician's Parkinson's disease symptom score on the UPDRS scale

Nerea Losada **Maitane Martinez**

Abstract

In this document we report our proposal for doing regression for the *motor* and *total UPDRS* scores from the 16 voice measures. We have implemented a *multi-layer perceptron* for this problem and compared it to two classifiers not based on NNs, which are *Linear Regression* and *Random Forest*. We have implemented the solution of this regression problem using mainly *scikit-learn*, *pandas* and *tensorflow* libraries that are available in python. In addition, we have read the dataset, prepared the data for regression and learned the model. For *multilayer-perceptron* we have divided the dataset in three sets, one for training, another one for validation and the last one for testing. From our results the best *MSE* has been produced by the *multi-layer perceptron* .

Contents

1 Description of the problem

This dataset is composed of a range of biomedical voice measurements from 42 people with early-stage Parkinson's disease recruited to a six-month trial of a telemonitoring device for remote symptom progression monitoring. The recordings were automatically captured in the patient's homes.

Columns in the table contain subject number, subject age, subject gender, time interval from baseline recruitment date, motor UPDRS, total UPDRS, and 16 biomedical voice measures. Each row corresponds to one of 5,875 voice recording from these individuals. The main aim of the data is to predict both the motor and total UPDRS scores ('motor_UPDRS' and 'total_UPDRS') from the 16 voice measures, so we have to simultaneously predicted two features. For this we have used different regression algorithms. The first two aren't based on *Neural Networks* (*Linear Regression* and *Random Forest*). The last one is *multi-layer perceptron*, implemented using *tensorflow*.

Attributes information:

- age- Subject age
- sex- Subject gender '0' - male, '1' - female
- test_time- Time since recruitment into the trial. The integer part is the number of days since recruitment.
- motor_UPDRS- Clinician's motor UPDRS score, linearly interpolated
- total_UPDRS- Clinician's total UPDRS score, linearly interpolated
- Jitter(%),Jitter(Abs),Jitter:RAP,Jitter:PPQ5,Jitter:DDP - Several measures of variation in fundamental frequency
- Shimmer,Shimmer(dB),Shimmer:APQ3,Shimmer:APQ5,Shimmer:APQ11,Shimmer:DDA - Several measures of variation in amplitude
- NHR,HNR - Two measures of ratio of noise to tonal components in the voice
- RPDE - A nonlinear dynamical complexity measure
- DFA - Signal fractal scaling exponent
- PPE - A nonlinear measure of fundamental frequency variation

2 Description of our approach

We organized the implementation of the project according to the following tasks:

1. Doing regression of the motor and total UPDRS scores from the 16 voice measures of the dataset.
2. Implement a *multi-layer perceptron* for this problem and compare it to one or more classifiers not based on NNs.
3. Implement the solution of the regression problem (the two regression problems simultaneously).
4. Evaluate and discuss the results of the regressor.

To complete our tasks, we follow some steps that are implemented in the notebook and explained in this report:

1. The reading of the dataset.
2. The preprocessing to get the data we need for the problem.
3. Dividing the dataset.
4. Implementation of a solution for the regression problem using regressors not based on NNs: *Linear Regression* and *Random Forest*.
5. Implementation of a *multi-layer perceptron*.
6. Definition of the architecture of the NN

7. Placeholders for the input data.
8. Initialization of weights and bias.
9. Optimization.
10. Creation of the model.
11. Validating the model.
12. Visualization of the results.

2.1 Reading the data

We have used the *panda* library to read the data. First of all, we define the names of the 22 attributes mentioned in *Section 1, Description of the problem*. Then, we read the data, that is in a *.data* file. To do so, we use the *read_table* function [1], which automatically gets all the information.

2.2 Preprocessing

We make a copy of the table, in order not to lose the information. After that, we drop from the data set those features we are not going to use, which are all except the 16 voice measures. Apart from that, we create another table in which we save the attributes we want to predict: *motor UPDRS* and *total UPDRS*. We obtain those features by using the function *concat*[2] to concatenate both columns.

2.3 Dividing the data set

We first divide the data set in two sets to compute the regressors not based in NNs: the training data and the test data. The parameter *n_samples* refers to the half of the samples. This way, we have the half of the samples for training and the other half for testing.

2.4 Regression problem

In accordance with our task, we have chosen three regression algorithms that are relevant for the type of regression problem, this way we can compare the *Mean Squared Error* got by each of them.

We have chosen *Linear Regression* and *Random Forest* that are not based on NNs and *multi-layer perceptron*.

2.4.1 Linear Regression

We use *Multiple Linear Regression (MLR)*, due to having multiple variables to predict the output of another variable. This algorithm, also known simply as multiple regression, is a statistical technique that uses several explanatory variables to predict the outcome of a response variable. The goal of *Multiple Linear Regression (MLR)* is to model the linear relationship between the independent variables and dependent variable.

We use the *sklearn.linear_model* [3] to obtain the model of this algorithm. Then we fit the training data with the features and the correspondent output. After training the model, we make the predictions using the test data and we evaluate the model computing *MSE*. The result we get for this function is 133, which is less than the *MSE* got with *Random Forest*.

2.4.2 Random Forest

Random Forest is a *Supervised Learning* algorithm which uses ensemble learning method for classification and regression. The trees in *Random Forest* are run in parallel and there is no interaction between these trees while building them, so it is a good algorithm for our problem. For regression, which is what we use, it operates by constructing a multitude of decision trees at training time and outputting the mean prediction of the individual trees.

We use the *sklearn.ensemble* [4] to obtain the model of this algorithm. The process to build this model is the same as in the previous case: we fit the training data with the features and the correspondent output. After training the model, we make the predictions using the test data and we evaluate the model computing *MSE*. In this case we get the highest *MSE* of the three algorithms.

2.4.3 Multi-layer Perceptron

Comparing to *Linear Regression* (and the linear network with no hidden layers) have a closed form solution. We can compute the optimal model directly and efficiently. Once we add an activation function, which in our case is *ReLU*, and possibly hidden layers, we cannot compute an optimal model directly anymore, so we are forced to use an iterative solution: an algorithm that goes through steps, usually improving the model with each step. There are no guarantees that the process will converge, or that you'll find the best model, which is exactly what happens: our model oscillates.

To compute this solution based on NNs, we have divided the data in three sets: train, validation, and test. For training we use half of the samples, for validation the last 1000 samples, and for testing the remaining samples. We use some samples to validate the model, since it learns too much for the train set, it is not able to generalize for the test set, and there is one iteration from which training loss decreases while test loss increases), this is, to avoid overfitting. So we use the validation set to decide when to stop the iterations.

We use *TensorFlow* to build the model and make the predictions. *TensorFlow* is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets pushing the state-of-the-art in ML and easily building and deploying ML powered applications. Clears the default graph stack and resets the global default graph.

We start by using the *reset_default_graph* function that clears the default graph stack and resets the global default graph, and defining the *mini_batch_size*. This parameter is used to split the training data into small batches. In our model we chose 50, according to the amount of samples. Nevertheless, we have tried different values to analyse the oscillating effect we have mentioned.

Then we define the architecture of the model, which consists of 16 neurons for the input layer (the number of the features we have), two hidden layers, the first one with 12 neurons and the second one with 6, and an output layer with 2 neurons, the number of features we want to predict.

We start the model construction by defining initialization variables. *num_output* is the number of attributes we want to predict, *motor_UPDRS* and *total_UPDRS*. *num_features* is the number of features we use for the prediction, this is, the 16 voice measures. *train_n_samples* is the number of samples in the training set, which are 1937. *num_layer_0* and *num_layer_1* are the number of neurons each of the hidden layers have.

After defining the initialization variables, we define a placeholder for the input and the output variables. A placeholder is simply a variable that we will assign data to at a later date. It allows us to create our operations and build our computation graph, without needing the data.

The hidden layers require weights and biases and they need to be initialized with a random normal distribution with zero mean and small variance (1/square root of the number of features). A weight represents the strength of the connection between units. Bias allows us to make sure that even when all the inputs are none (all 0's) there's gonna be an activation in the neuron. What we use to define those in the model is the*tf.Variable* from *TensorFlow*.

After that, we start implementing the graph calculation to develop our 16(Input)-12(Hidden layer 1)-6(Hidden layer 2)-2(Output) model. We multiply the input of each layer with its respective weights and add the bias. Then we need to add an activation function; we will use *ReLU* for hidden layers . Activation functions are used to introduce non-linearity to neural networks.*ReLu* puts the negative values to 0.

We need to define a loss function to optimize our weights and biases, and we will use *Mean Squared Error* for the predicted and correct labels. We use the *reduce_mean* in order to compute the mean of elements across dimensions of a tensor.

We define an optimizer and the learning rate for our network to optimize weights and biases on our given loss function. We will use 0.001 for variable learning rate. The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. We decided that change to be very small to understand gradually. For optimizing, we are going to use Adam optimizer. Adam is an optimization algorithm that can used instead of the classical stochastic gradient descent procedure to update network weights iter-

ative based in training data. Stochastic gradient descent is a method to find the optimal parameter configuration for a machine learning algorithm.

We can now start training our model with the training set and evaluating it with the test set. We use *initialize_variables* to implicitly passing *tf.global_variables* [5] as a *var_list*. The number of training epochs is 500000. One epoch is one forward pass and one backward pass of all the training examples. So the model is going to loop a maximum of 500000 times. The number of *display_step* is 1000. We use this variable to compute it every 1000 steps. *n_batch* variable is how many batch we have split. We create *list* structure arrays to save the errors of train, validation and test. We use those to plot the *MSE* of each set.

We need to create a session for the graph using *Session* function [6]. The session will also allocate memory to store the current value of the variable. Then, we run the *init* to initialize the session.

We decided to stop the model from learning when overfitting occurs. For that aim, we compare the new validation loss got in each iteration with the minimum one. If it is lower, we update it and we save the epoch loop number and the error got. If it is higher, we compared the new validation loss with the previous one. If the loss increases, we increment the counter *max*. When the counter is higher than 10, this is, the error increases for 10 consecutive iterations, we decide that the model is not learning anymore, so we save the epoch number in which it has started to increase and stop the training process.

We use *plot* function to draw the error for the three sets along the epochs.

2.5 Validation

We use *Mean Squared Error* [7] to compute the accuracy of the previous explained models. The *MSE* is the average of the squared error that is used as the loss function for least squares regression.. *MSE* is a risk function, corresponding to the expected value of the squared error loss. The fact that *MSE* is almost always strictly positive (and not zero) is because of randomness or because the estimator does not account for information that could produce a more accurate estimate.

3 Implementation

All the project steps were implemented in Python. We used *pandas* for reading and preprocessing the dataset, and *scikit-learn* and *TensorFlow* for the regression tasks. We illustrate how the implementation works in the Python notebook *ParkinsonMotorRegression.ipynb*.

4 Result

This are the *Mean Squared Error* we obtained:

1. *Linear Regression* : 133.57
2. *Random Forest*: 146.215
3. *Multi-layer perceptron* : 125.25

The best result is achieved by the *multi-layer perceptron* algorithm, but there is not a big difference with the others. The *Linear Regression* is the second best one, so from those results we can conclude that the data has a linear shape. An advantage of the *Linear Regression* is that you can compute the optimal model directly and efficiently.

Furthermore, in the notebook we can see the graphic obtained in the *multi layer perceptron* with a mini_batch_size of 50. In it, we can observe some oscillation. We can change it by using another optimizer or changing the *mini_batch_size* . We have tried increasing and decreasing the*mini_batch_size*, but we got more exaggerated oscillation effect, as we can see in the following images.

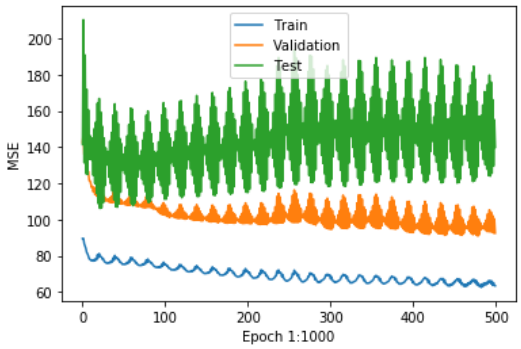

Figure 1: mini_batch_size=25

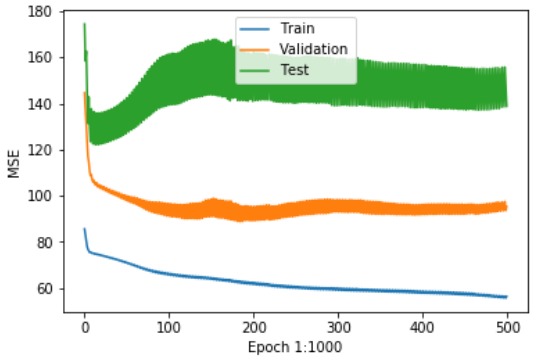

Figure 2: mini_batch_size=100

5 Conclusion

In our project we have applied the *Linear Regression*, *Random Forest* and *multi-layer Perceptron* regression algorithms to the *Parkinson Motor dataset*, introduced in [8]. We have computed the *Mean Squared Error* of those regressors and observed that *multi-layer perceptron* produces the best result. We have also observed, from the analysis of the graphics, that we have to deal with overfitting. A way to prevent overfitting is to add some drop out after each hidden layer. Dropout is an essential concept in creating redundancies in our network which leads to better generalization. Therefore, we have also run the training algorithm several times and observe that the results vary a lot, which means that the dependency of the model on weight initialization is high.

References

[1] Pandas read_table [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_table.html].

[2] pandas concat [https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.concat.html].

[3] LinearRegression [https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.linearregression.html].

[4] Random Forest classifier [https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.randomforestclassifier.html].

[5] Tensorflor global_variable [https://www.tensorflow.org/guide/variable].

[6] Tensorflor session [https://www.tensorflow.org/api_docs/java/reference/org/tensorflow/session].

[7] Mean Squared Error [https://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html].

[8] PE McSharry LO Ramig A Tsanas, MA Little. Accurate telemonitoring of parkinson.s disease progression by non-invasive speech tests. Technical report, IEEE Transactions on Biomedical Engineering, 2009.