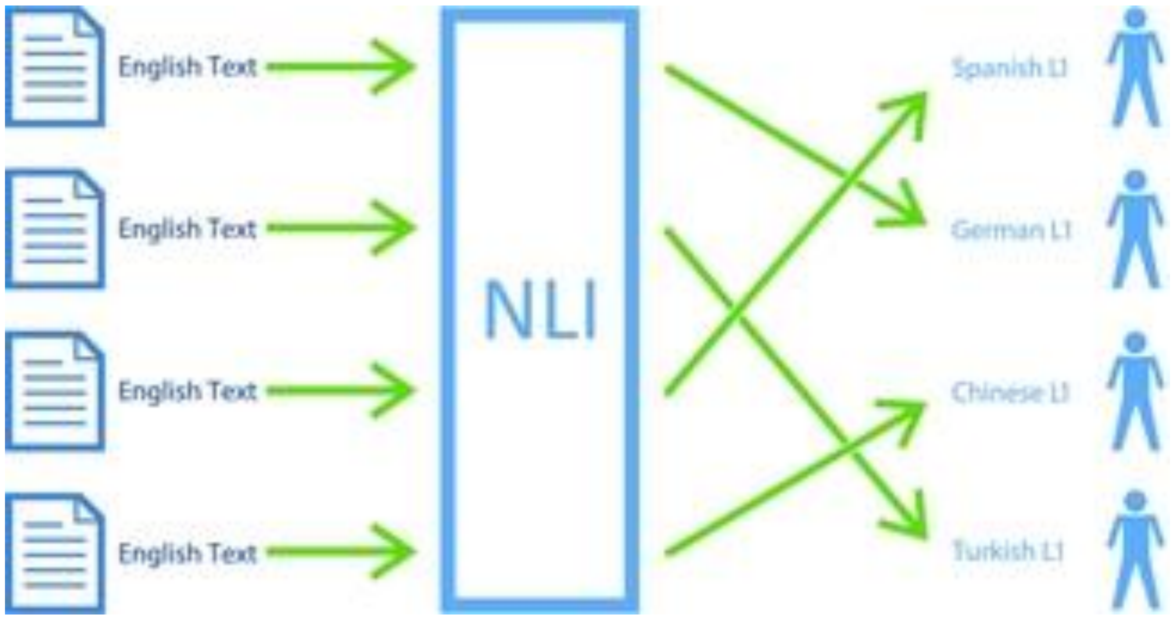
Native Language Identification



Author: Julen Arizaga Echebarria

Contents

Overview

Through this work it will be explained how to build a model in order to do native language identification, using feature vectors derived from BERT. In order to obtain the best possible model, different models will be used, and different accuracy measures too, in order to see which performs best for the given data.

The data used consists of the following:

- lang_id_train.csv: 6000 texts with labels used to train the model
- lang_id_eval.csv: 2000 texts with labels used to tune the hyperparameters of the model
- lang_id_test.csv: 2000 texts with labels for the evaluation

All of those files have 10 different categorical targets (totally balanced), and are the following: 'Japanese', 'Korean', 'Mandarin', 'Vietnamese', 'Thai', 'Spanish', 'Arabic', 'Cantonese', 'Polish', and 'Russian'.

Steps to reproduce the results

Data needed

In order to build the model and have the same results that will be provided, the following data and scripts are needed:

- **BERT program:** This is the program, that has multiple python scripts, needed in order to obtain the feature vectors out of the texts that are going to be used for the native language identification.

- **Preprocessed data for BERT:** As getting the preprocessed data for BERT may take a lot of time and effort, some preprocessed data provided by the creators of BERT is going to be used.

- **Training, evaluation and test data:** Those are some csv files, which have the native language in one column and the text in another column.

- **BERT feature vector obtention script:** This script uses the BERT program in order to obtain the feature vectors needed for the modeling part. Its name is 'run_bert_fv.sh'.

- **Preprocessing python script:** The BERT python scripts need a .txt file with only text in each row, therefore a script to preprocess the data is needed. Its name is 'Preprocessing.ipynb'.

- **Model building python script:** This is the script that has been used in order to build different models and test them with different accuracy metrics. Its name is 'native language identification.ipynb.

Preprocessing

As said before, some preprocessing is needed in order to obtain the .txt files that are going to be used by the BERT program from the .csv files provided. In order to obtain them, the previously mentioned "**Preprocessing**" is provided.

Note that, in order to run this script, the directory of the .csv files must be specified in the variable "INPUT_DATA", which will change for different computers. In this case the output file will be the same as the input file.

Obtaining Feature Vectors

The 2nd step is to obtain the feature vectors (that are going to be used to fit the models) from the preprocessed .csv files, and to do this the Bert program has to be used. Several parameters and text files must be sent to the "extract_features.py" script of the program, and to do so a shell script has been used, the "run.sh" script.

In order to run this script in any computer, four directories must be specified:

- BERT_BASE_DIR: directory of the BERT program.
- BERT_DATA_DIR: directory of the pre-trained data.
- INPUT_DIR: directory of the input data. In our case this will contain the train, evaluation and test sets, which will be transformed from text into vectors. Note that this data must be preprocessed as said before, so this directory must be the one used in the preprocessing step.
- OUTPUT_DIR: directory of the output data, the vectors. Those vectors are the ones used in the next step of building a model.

Running the python scripts

In order to run those scripts and obtain the results that will be later explained, Python 3.7.4 should be used. Note also that this has been run in Windows 10, because if run in another operating system, a different result could be obtained for the neural network part.

As said before, three python scripts are provided, and in order to run them adequately only the following paths should be changed:

- **ORIGINAL_DATA_DIR:** Directory of the original training, eval and test data
- **BERT_FEATURE_DIR:** Directory of the obtained BERT feature vectors
- **PRINT_RESULTS:** Directory where the results (images and csv file) should be saved

Building Models

Introduction

Predicting the native language of a person just by the way he writes in English is no easy task, therefore different algorithms will be tried in order to see which one gives the maximum possible accuracy. At first glance, as it is a difficult task, a complex algorithm like a neural network is expected to behave more accurately than others such as logistic regression or support vector machine.

Several algorithms will be tried, and for each algorithm a grid search will be done, in order to tune the hyperparameters to select the best model for our data. The grid search will be done using the accuracy over the evaluation set as metric. Finally, the accuracy of the test set will be calculated, and this is the number that will tell us how well the algorithm behaves.

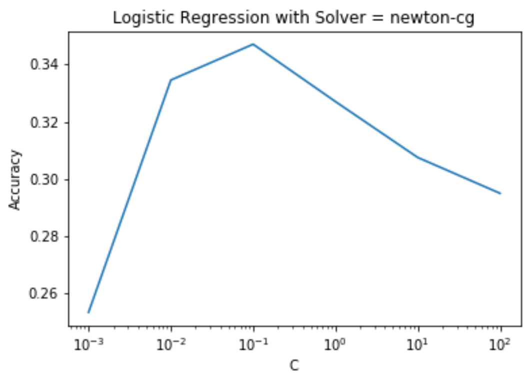
In the end, the best model will be taken and its results will be studied with different metrics in order to see which are the most frequent errors.

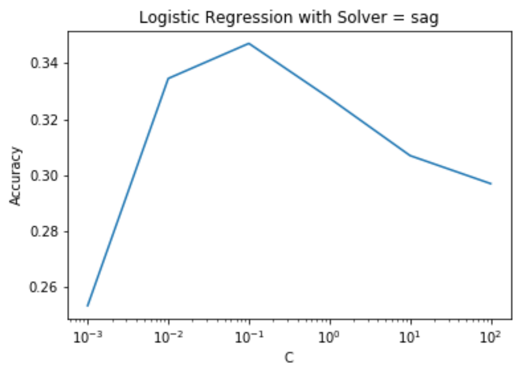
Logistic Regression

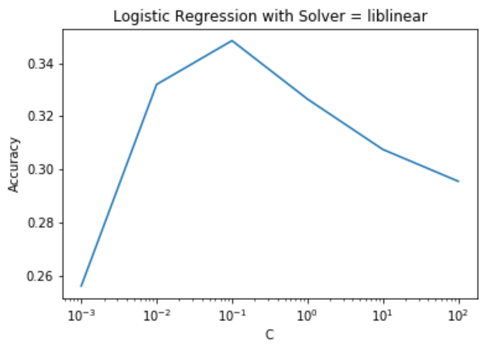
For the Logistic Regression algorithm those are the hyperparameters that have been tuned:

- Solver: liblinear, sag, newton-cg
- Inverse Regularization C: search from 0.001 to 100

The first grid search returns the following result:



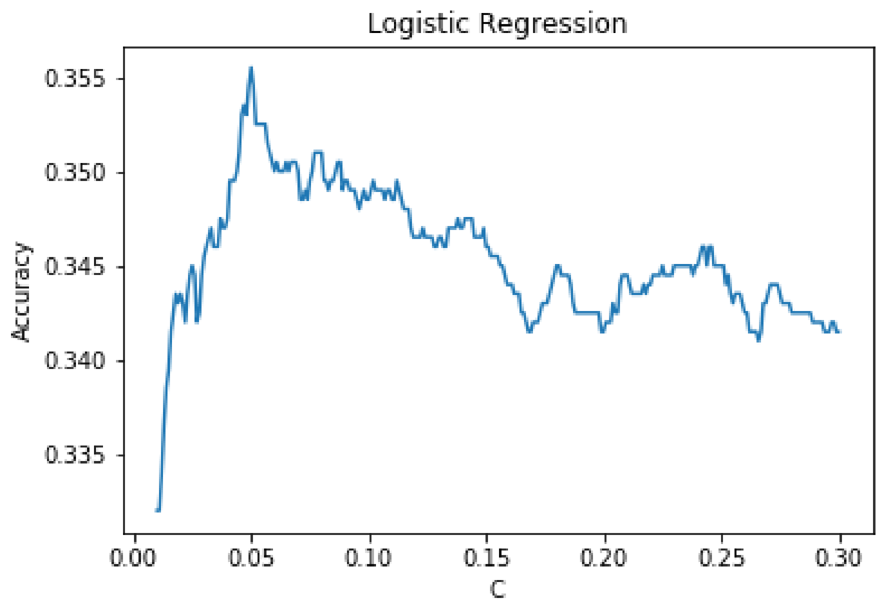




As it can be seen in the images, the maximum accuracy is likely to be between C = [0.01, 1], and the best solver seemes to be ‘liblinear’, so a further search has been developed in this C range and for this solver.

It has to be said that all the solvers seemed to have more or less the same accuracy for the same C, but ‘sag’ needed about 700 iterations to converge and ‘newton-cg’ had a little bit less accuracy.

Finally, this is the graph obtained for the ‘liblinear’ solver and C = [0.01, 1] range:



As it can bee seen, the maximum accuracy has been obtained for C = 0.05 which in this case is 0.355. Therefore the best obtained model for the Logistic Regression algorithm is the following:

Best Model:

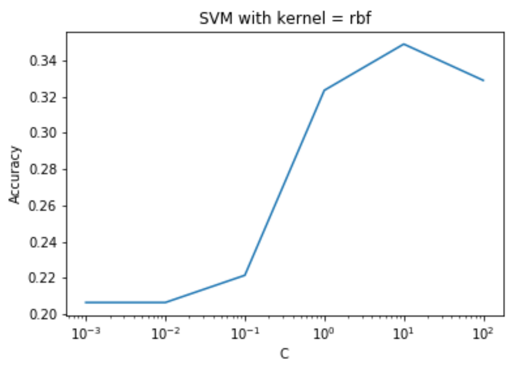
- Solver: 'liblinear'
- C = 0.05
- Penalty = 'l2'
- Accuracy (for the test set) = 0.353
- Time taken = 0.1153 minutes

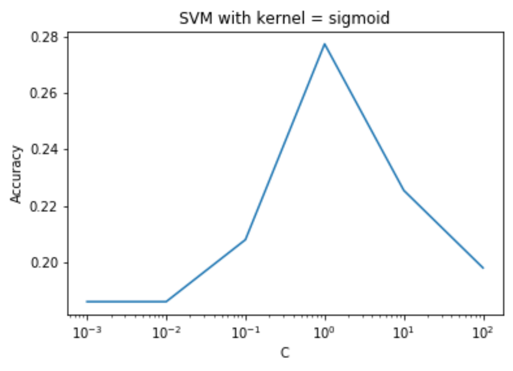
Support Vector Machine

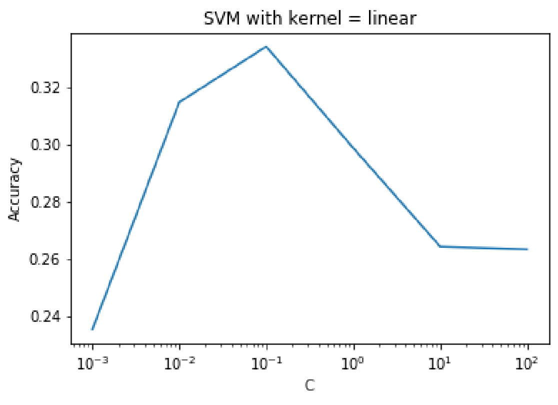
For the Support Vector Machine algorithm those are the hyperparameters that have been tuned:

- Kernel: rbf, linear, sigmoid
- Inverse Regularization C: search from 0.001 to 100

The first grid search returns the following result:





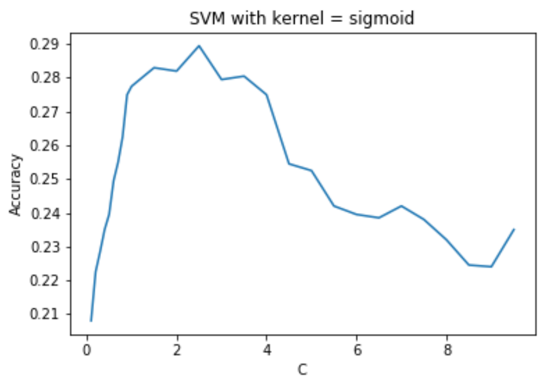


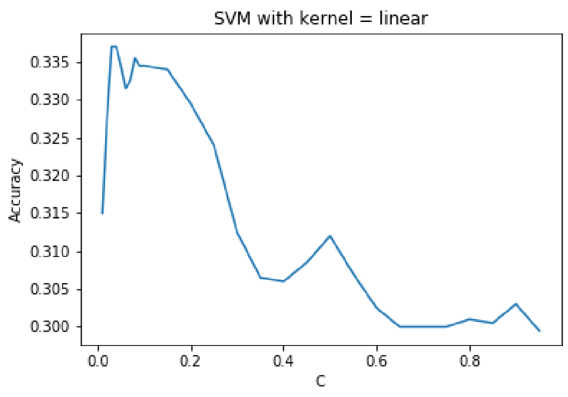
As it can be seen in those images, the best kernel seems to be the 'rbf', but as there is not enough evidence to say so and the values of C are few and sparse for every kernel, a deeper search is going to be developed for each kernel where it seems to be the maximum.

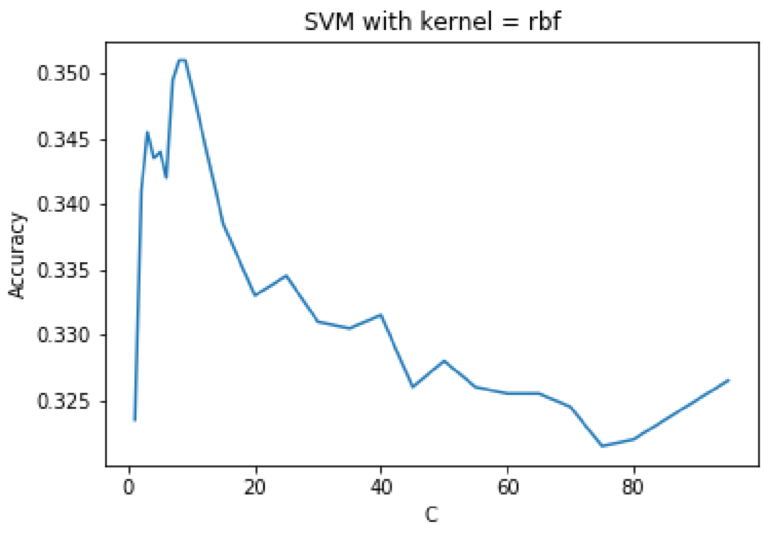
The search is going to be the following:

- Kernel 'rbf': C between [1,100]
- Kernel 'sigmoid': C between [0.1,10]
- Kernel 'linear': C between [0.01, 1]

This second grid search returns the following result:

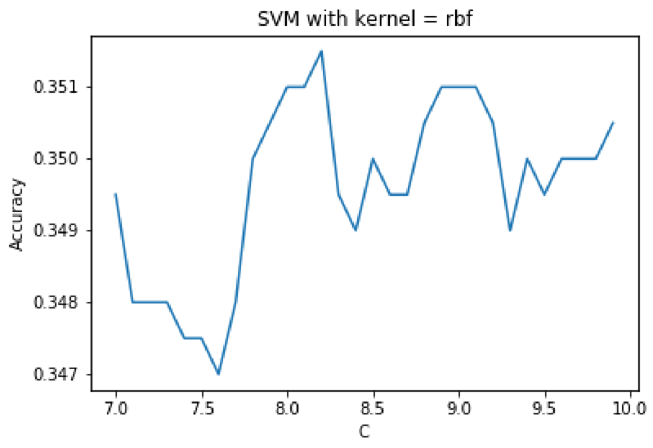






As it can be seen in those pictures, the maximum accuracy seems to be obtained for the 'rbf' kernel between 7 and 10, therefore an even more precise search is going to be done, in order to obtain the best C.

This third grid search returns the following result:



Finally the highest accuracy of 0.3515 has been obtained for C = 8.2. The resume of the best found model of for the Support Vector Machine algorithm is the following:

<u>Best Model:</u>

- Kernel: 'rbf'
- Decision_function_shape = 'ovr'
- Gamma='scale'
- C = 8.2
- Accuracy (for the test set) = 0.346
- Time taken = 1.46 minutes

Neural Networks

Nowadays Deep Learning is the state of the art for many NLP problems, so the Neural Networks are expected to have a higher accuracy than the previously seen algorithms. One of the best approaches actually used for text classification as well as other problems in NLP is to use LSTM neural networks, but as this is no easy task and the BERT feature vectors would need to be extracted differently, the simple neural network approach (with dense layers) will be used.

Simple Neural Networks

There are two widely used libraries in Python to build the simple neural networks model, and in this project, both are going to be used to see the difference between them. Those two libraries are 'Sklearn' and 'Keras'.

The grid search in the Neural Networks is a much more complicated problem than in the previous algorithms because of the following:

- The accuracy does not follow a specific trend when the number of layers or the number of neurons is changed.
- The more complex the model, the more complex the optimization function gets, and as it is convex, there are more possibilities of reaching a local minimum point, and a bad one. This would result in an early convergence but a high loss and bad accuracy.
- The solver plays a big role here. Different solvers and different steps will converge to different local minimum, and there is no way of deciding which one is going to be the best for the given network.

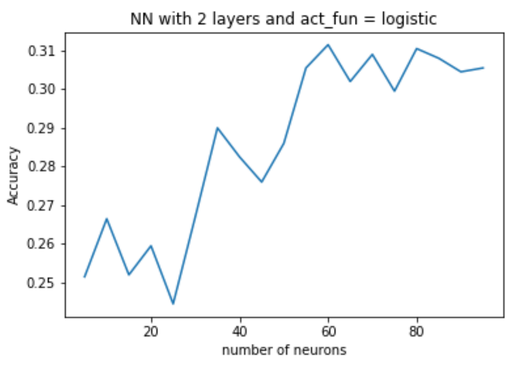
Because of all of this, in order to find the best model, a wide and difficult grid search would have to be done, which would take too much time. In order to go through different types of Neural Networks and see how they behave for our data, a limited grid search is going to been done.

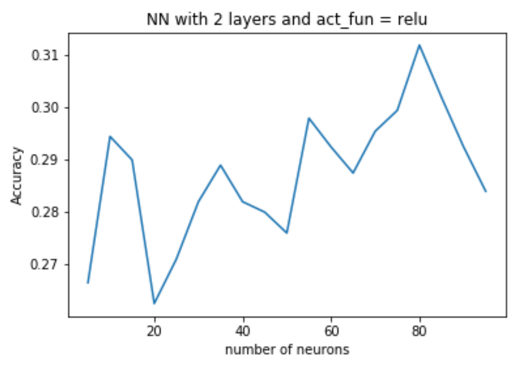
Sklearn library

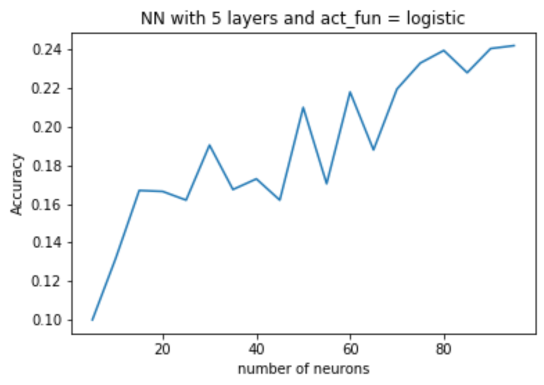
For the Neural Networks algorithm with the Sklearn library, those are the hyperparameters that have been tuned:

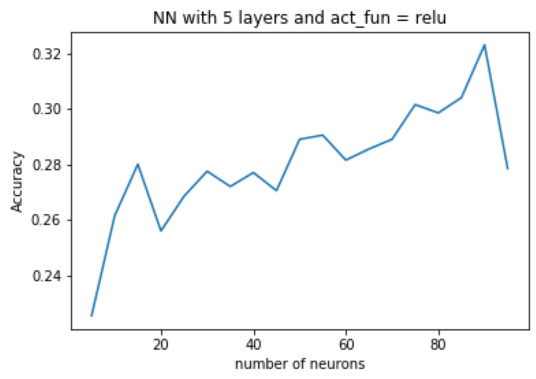
- Activation function: logistic, relu
- Hidden layers: search between [2,15]
- Number of neurons: search between [5,100]

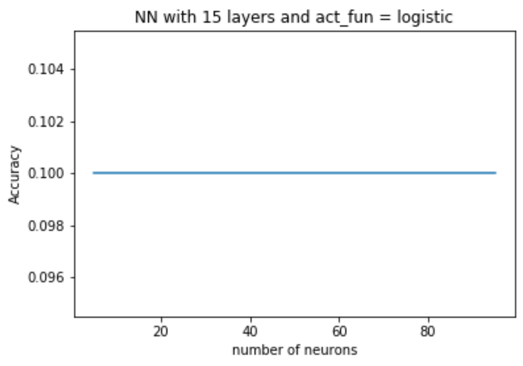
The first grid search returns the following:

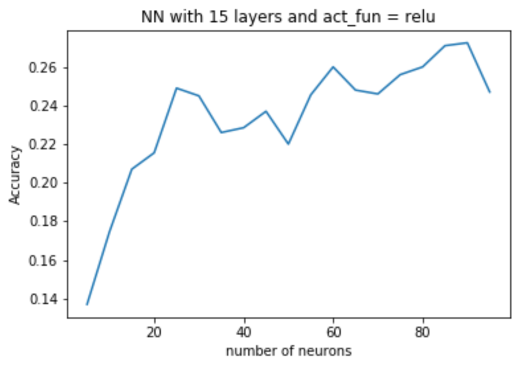








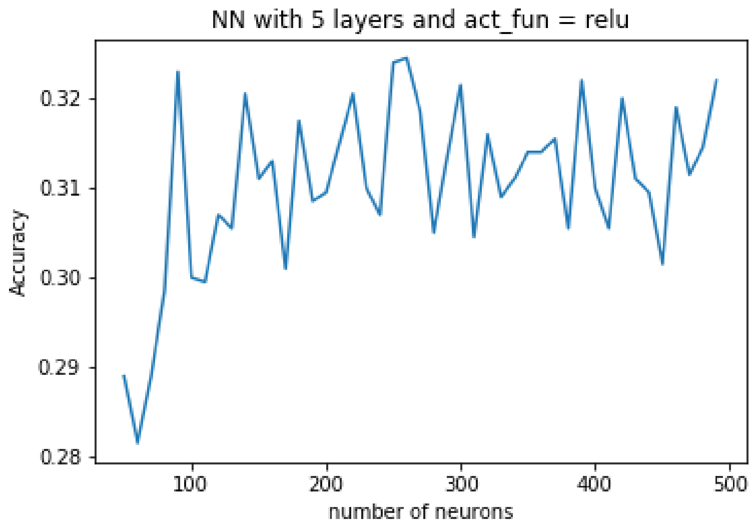




As it can be seen in the images, in general it seems that fewer layers and more neurons get a better result. This seems to be the general trend, but it is not always upward or downward, and for the 15 layers case with the 'logistic' activation function, there seems to be some kind of problem with the solver (maybe early local minimum), because it converges very fast but gets a bad accuracy (the same as choosing the language randomly). Because of all of this, a further study will be developed for 5 hidden layers and 'relu' activation function, as those are the parameters that get the best accuracy for [5,100] number of neurons.

The second grid search's parameters:

- Activation function: relu
- Hidden layers: 5
- Number of neurons: search between [50,500]



As it can be seen in the image, there seems not to be a specific trend when the number of neurons is higher than 200. As said before, this could be due to many factors, so the best model from here will be taken, in this case the one with 260 neurons per layer.

Best Model:

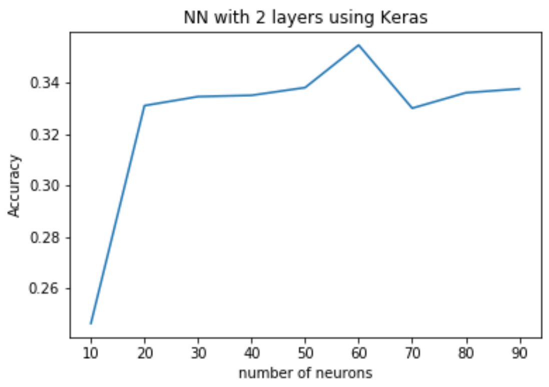
- Solver: ‘adam’
- Activation function: ‘relu’
- Number of hidden layers: 5
- Number of neurons per layer: 260
- Accuracy (for the test set) = 0.325
- Time taken = 1.05 minutes

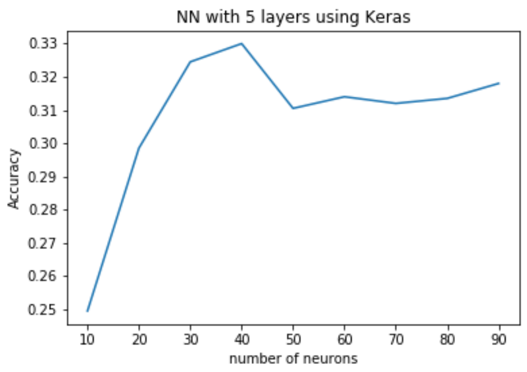
Keras library

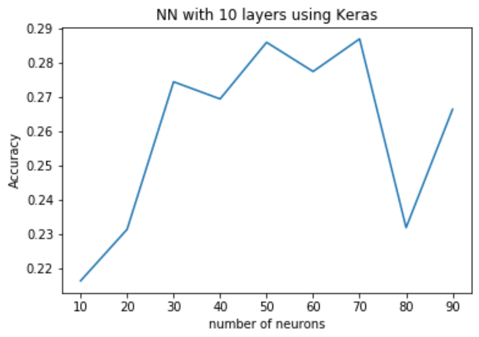
For the Neural Networks algorithm with the Keras library, those are the hyperparameters that have been tuned:

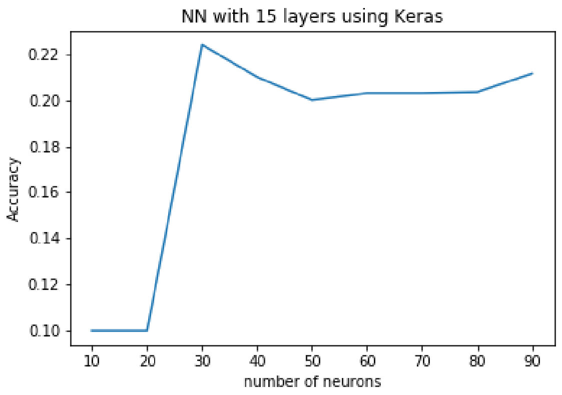
- Hidden layers: search between [2,15]
- Number of neurons: search between [10,100]

The activation function has been set to 'relu' as it had a better performance with the previously tested neural network. The first grid search returns the following:









In this case again, the neural networks with fewer layers obtained better performance than the ones with more layers, and the accuracy seems to be higher for more neurons per hidden layer, although this trend cannot be confirmed from those graphs. The following has to be said about those graphs:

The number of epochs and the batch size used for those simulations has been 15 and 32, which has not changed with the number of layers or number of neurons. Looking at the convergence of the simulation as it goes, it can be seen that the model with fewest layers gets easily overfitted with more than 10 epochs (in the end, with 15 epochs, numbers like training accuracy 0.5 and validation accuracy 0.3 are obtained), but the models with more layers would increment their performance with more epochs.

Therefore, the number of epochs as well as the batch size should be changed for every layer size and number of neurons, in order to get the best possible accuracy for the validation set.

The best model obtained from here is the following:

Best Model:

- Solver: ‘adam’
- Activation function: ‘relu’
- Number of epochs: 15
- Batch size: 32
- Number of hidden layers: 2
- Number of neurons per layer: 60
- Accuracy (for the test set) = 0.3404
- Time taken = 0.4425 minutes

Conclusions

Best model

From the three algorithms used, the best model found has been the one using the Logistic Regression algorithm, with an accuracy of 0.353. Its parameters are the following:

Best model's parameters:

- Algorithm: Logistic Regression
- Solver: 'liblinear'
- C = 0.05
- Penalty = 'l2'
- Accuracy (for the test set) = 0.353
- Time taken = 0.1153 minutes

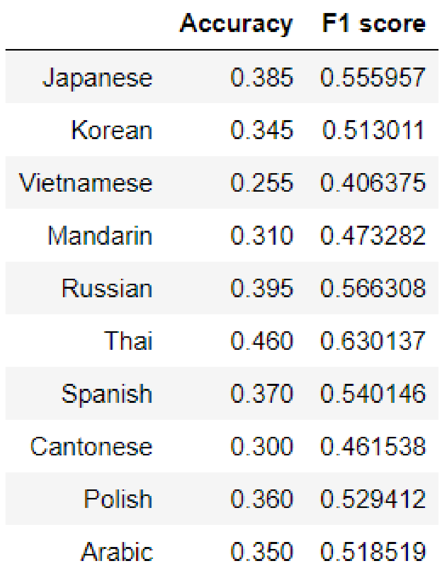
As it can be seen this simple algorithm, which takes much less time than other complex algorithms, has performed better for the dataset. Now that the best model has been decided, the overall performance, metrics by class and frequency of errors between classes will be commented.

Overall performance

As previously said, the overall performance of this model is 0.353 measured by the accuracy metric. As there are 10 classes, if the predicted class was chosen randomly the accuracy would be 0.1, so the performance is 3.53 times higher using the model than doing it randomly. It still doesn't seem very good to have 0.353 of accuracy, which means, that 64.7% of the predicted labels will be incorrect, but the difficulty of the classification should also be considered. One good way to decide if this model is good or bad would be to calculate the Cohen's kappa, because that way the accuracy of a professional human would be considered.

Metrics by class

The accuracy and F1 score for each class have been calculated, and they can be seen in the next table:



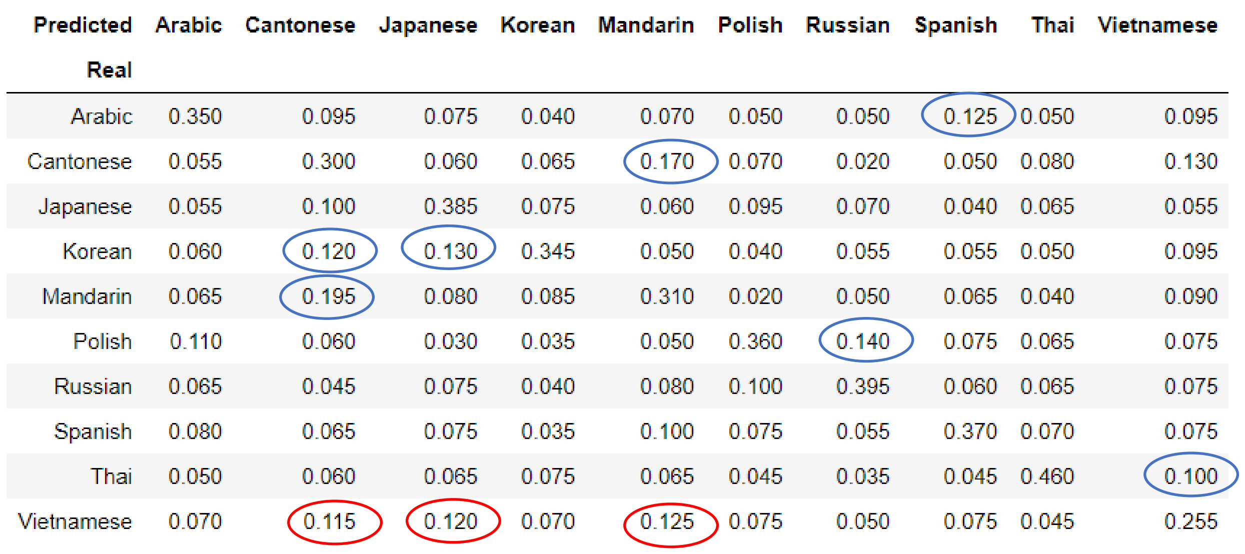
	Accuracy	F1 score
Japanese	0.385	0.555957
Korean	0.345	0.513011
Vietnamese	0.255	0.406375
Mandarin	0.310	0.473282
Russian	0.395	0.566308
Thai	0.460	0.630137
Spanish	0.370	0.540146
Cantonese	0.300	0.461538
Polish	0.360	0.529412
Arabic	0.350	0.518519

As it can be seen the best accuracy goes for the “Thai” language and the worst for “Vietnamese”. Those two cases are quite extreme compared to the other ones, all the other classes have the accuracy between 0.3 and 0.4.

It has to be said that the ranking of the languages according to the accuracy and according to the F1 score is somehow different for some languages, but the overall appearance would me the same.

Frequency of errors

In order to understand better why some of these errors happen, the normalized confusion matrix is going to be plotted and commented (normalized by the number of true labels per class, which in this case is 200 and equal).



Predicted	Arabic	Cantonese	Japanese	Korean	Mandarin	Polish	Russian	Spanish	Thai	Vietnamese
Real										
Arabic	0.350	0.095	0.075	0.040	0.070	0.050	0.050	0.125	0.050	0.095
Cantonese	0.055	0.300	0.060	0.065	0.170	0.070	0.020	0.050	0.080	0.130
Japanese	0.055	0.100	0.385	0.075	0.060	0.095	0.070	0.040	0.065	0.055
Korean	0.060	0.120	0.130	0.345	0.050	0.040	0.055	0.055	0.050	0.095
Mandarin	0.065	0.195	0.080	0.085	0.310	0.020	0.050	0.065	0.040	0.090
Polish	0.110	0.060	0.030	0.035	0.050	0.360	0.140	0.075	0.065	0.075
Russian	0.065	0.045	0.075	0.040	0.080	0.100	0.395	0.060	0.065	0.075
Spanish	0.080	0.065	0.075	0.035	0.100	0.075	0.055	0.370	0.070	0.075
Thai	0.050	0.060	0.065	0.075	0.065	0.045	0.035	0.045	0.460	0.100
Vietnamese	0.070	0.115	0.120	0.070	0.125	0.075	0.050	0.075	0.045	0.255

Vietnamese: As it can be seen in the table above, the program mistakes the Vietnamese native language with the Cantonese, Japanese and Mandarin too much, that's why the accuracy of this one is so low.

Thai: In the other hand, the Thai native language is mostly mistaken with the Vietnamese, but not very much with other languages, that's why it has the best accuracy.

Others: Other strong error relations that frequently happen are the following:

- Arabic mistaken as Spanish
- Cantonese mistaken as Mandarin
- Korean mistaken as Cantonese or Japanese
- Mandarin mistaken as Cantonese
- Polish mistaken as Russian

With these frequent errors in mind further parameters could be added to the model in order to separate better those highly mistaken languages.