Invictus Capital

Data Scientist Assessment

Language Identifier

Hannes Engelbrecht

May 2020

GENERAL OVERVIEW AND CONTENTS

This report details the work done for the first Data Scientist assessment – building a Language Identifier (LID) model that is able to discriminate between English, Afrikaans and Dutch phrases.

The following areas are discussed:

1. External Python **libraries** used in the implementation
2. A complete overview of the **Exploratory Data Analysis** (EDA), **data preparation** and **feature engineering**
3. A discussion on the chosen model's **architecture**
4. The model **training process**, including specific techniques and algorithms
5. **Testing** the model
6. Model **results**, overall and split by language
7. **Bonus questions**

The aforementioned steps were conducted in a Jupyter Notebook. In addition, the trained model and its parameters were saved separately and made available on Github (https://github.com/HannesEngel/Invictus_Capital_Language_Detection) for the project reviewer to reproduce the results. Instructions on how to do so can be found in the provided Github link's README.md file.

1. LIBRARIES

Below I have set out each of the external libraries, along with a brief overview of its main uses, utilised during the course of this project.

- NumPy – for working with arrays
- Pandas – for manipulating dataframes
- Matplotlib – for visualizing data
- Missingno – for visualizing missing data
- Regular expression (re) operations – this allows us to clean and modify the text data with relative ease
- Json – for loading the JSON model file (in which the trained model is stored)
- Scikit-Learn – this library is used for splitting the dataset into a train and test set and encoding the labels
- Tensorflow – open source machine learning (ML) platform, particularly useful for building more complex ML models
- Keras – neural network library (built on top of Tensorflow) with a high-level API. Keras was used to construct, train and test our model.

2. EXPLORATORY DATA ANALYSIS, DATA PREPARATION AND FEATURE ENGINEERING

Distribution of the data:

Upon first glance, we observe that the *lang_data.csv* file contains a total of 2839 rows. The phrases contained in this file are distributed among only 3 languages ('English', 'Afrikaans' and 'Nederlands'), which are consistently labelled. The original file is distributed among these 3 languages as follows (numbers are displayed as percentages of the total):

English	73.159563
Afrikaans	23.635083
Nederlands	3.205354

We immediately recognize the fact that we have an imbalanced dataset, and hence a class imbalance problem that will need to be addressed. This means that we have a dataset where there are many records of one or more classes in the dataset, but with very few records of one class (Dutch, in this case). This will be discussed further in the Model architecture and Bonus questions sections.

Missing values:

Next, we check the extent of missing data. There are 78 rows with any form of missing data. This amounts to 2.75% of the total dataset, which is a very small proportion. We also recognize that all of the rows with missing values only have a missing 'text' entry. I.e. in these 78 rows a language has been assigned to a blank/empty phrase. Options for imputing these missing phrases are few and could be time-consuming. Seeing that the number of missing values is small, it makes sense to rather omit them from the analysis and modelling stages completely. This was done using the '.dropna(how='any')' command. We therefore remain with 2761 rows with a distribution among languages as shown below. A last point to note on the missing values is that they are predominantly Afrikaans and Dutch.

English	74.429555
Afrikaans	23.143788
Nederlands	2.426657

Text pre-processing:

Taking a quick look at the 'text' column in our original dataset, we notice that the phrases contain special characters, for example hyphens and backslashes, and extra whitespace. For the purpose of this project these special characters and spaces are deemed to not have any predictive power when modelling the language of the phrase in question. The special characters are therefore omitted from the text using the regular expression library 're' mentioned in section 1 above. It could well be that certain languages make use of more special characters than others. It could also be that some languages use different special characters than others. If this were the case, the presence or otherwise of special characters could help predict the language of a phrase. This is not considered any further in this project and may be regarded as a potential area of improvement.

Similarly, the text contains both uppercase and lowercase letters. For the purpose of this project uppercase letters are deemed to not have any predictive power when

modelling the language of the phrase in question. All uppercase letters are therefore converted to lowercase letters. Again, it could be argued that some languages inherently have more/fewer uppercase letters. In this case the presence or number of uppercase letters in a phrase could be a predictor for the phrase's language. This is not considered any further in this project and may be regarded as a potential area of improvement.

Feature engineering:

One of the more important and time-consuming areas of using text as input to a model is to convert the text into a format that the model can understand. There are many ways of doing so and no single correct approach. The chosen method does however have a significant impact on the model's performance and it's worth noting that different methods of pre-processing the text is likely to lead to completely different model results.

The text feature engineering approach taken in this project is a simple one and is described in more detail below:

> Step 1: After the text is cleaned, we create a vocabulary of all words in the 'text' column of the remaining dataset.
> Step 2: Add '<UNK>' to the vocabulary. This is to ensure that the any new, unknown phrases that the model has not seen before don't cause the model to break (when making predictions). We end up with 4444 words in our vocabulary, including '<UNK>'.
> Step 3: Assign unique integers to each of the words in the resulting vocabulary, i.e. convert the words to numbers
> Step 4: Convert each phrase into an array of integers, with each word in that phrase converted to its vocabulary integer
> Step 5: Pad each of the resulting numeric arrays to a fixed length. During our EDA we observed that the longest phrase contains 65 words. We choose 200 as the length of each padded array to ensure that longer phrases (for which the model needs to predict a language) are catered for.

The arrays resulting from step 5 above were used as the independent variables/input to the model.

It is also necessary to encode the dependent variable ('language'). This is because the ML model needs to be fed numeric labels, not alphanumeric labels. Sklearn's OneHotEncoder is used as opposed to Sklearn's LabelEncoder to ensure that no order/hierarchy is implied by assigning larger/smaller integers to different languages.

A common practice in Natural Language Processing (NLP) is to stem/normalise the words. For example, after stemming both 'training' and 'trains' would be converted to 'train'. This way, both the sentences 'he is training the model' and 'he trains the model' would (correctly) have the same meaning. Another consideration is that of stopwords. For example, in English the word 'it' is considered a stopword. These stopwords are often removed in sentiment analysis tasks and are believed to potentially improve such models' performance. We do, however, not remove the stopwords. This is because the presence of the word 'it' or 'the' could indicate to the

model that the phrase is English (and not Afrikaans or Dutch), regardless of the fact that it's a stopword.

3. MODEL ARCHITECTURE

There are many viable model architectures for the task at hand. We make use of a Long Short-term Memory (LSTM) Recurrent Neural Network (RNN) to model the language of a phrase. This is essentially just a standard neural network that takes as input, in addition to the input from that time step, a hidden state from the previous time step. The hidden state acts as the neural networks memory. It holds information on previous data the network has seen before.

While RNNs learn similar to standard neural networks while training, they also remember things learnt from prior input(s) while generating output(s). LSTMs are a special kind of RNN, capable of learning long-term dependencies and are particularly useful for text classification.

For this model, we make use of an LSTM RNN with 4 layers. These layers are arranged as follows (in order):

1. Embedding layer: The Embedding layer is used to create word vectors for incoming words. It sits between the input and the LSTM layer, i.e. the output of the Embedding layer is the input to the LSTM layer. The Embedding layer is initialized with random weights and will learn an embedding for all of the words in the training dataset. The Embedding layer is defined as the first hidden layer of a network. It must specify 3 arguments:

    - **input_dim:** This is the size of the vocabulary in the text data. In our case this is 4444.
    - **output_dim**: This is the size of the vector space in which words will be embedded. It defines the size of the output vectors from this layer for each word. In our case this is set to 300.
    - **input_length**: This is the length of input sequences, as you would define for any input layer of a Keras model. In our case this number is set to 200.
2. LSTM layer: For this layer, 4 arguments are specified:

    - **units**: These are 'memory units'. Dimensionality of the output space. For our model it has been set to 256. This value is arbitrary and can easily be 'tuned', i.e. we can iterate over a range of values and find the value that maximises model performance.
    - **return_sequences**: Whether to return the last output. This is set to True, which allows us to access the hidden state output for each time step.
    - **dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the inputs. In our case it has been set to 0.5. One could argue that a dropout of 0.5 is relatively high. However, there is some evidence to suggest that a dropout rate in the 0.4 – 0.6 range results in optimum model performance (https://machinelearningmastery.com/use-dropout-lstm-networks-time-series-forecasting/). I have therefore chosen the midpoint of

this range and set the dropout to 0.5. This parameter can easily be ‘tuned’ to find the value that supports optimal model performance.

- **recurrent_dropout**: Float between 0 and 1. Fraction of the units to drop for the linear transformation of the recurrent state. In our case it has been set to 0.5, similar to the **dropout** parameter above. This parameter could be independently ‘tuned’ as well.

3. LSTM layer: This layer is similar to the above LSTM layer, with the exception that the **return_sequences** argument is set to False (default).

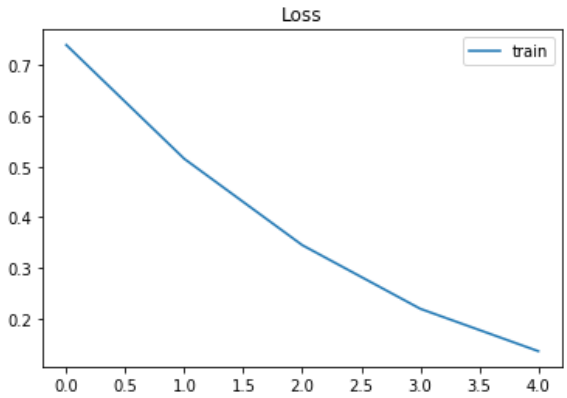
4. Dense layer: This is a fully connected layer (connected to each node in the previous layer) and is used for outputting a prediction. This layer provides a probability of the text phrase belonging to each of the 3 languages. In this layer we also specify an **activation function** that defines the format that the prediction takes on. For multiclass problems such as this one we use the ‘softmax’ activation function. This activation assumes a one-hot encoded output pattern, which is what we have done.

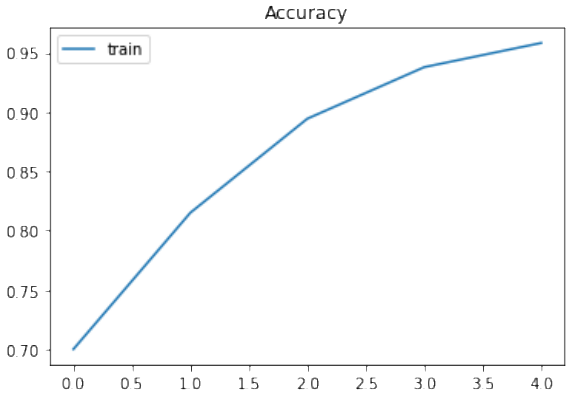
Other features of the architecture that we have specified include:

- Loss – for this problem we use the **categorical_crossentropy** loss, which is the default loss function to use for multi-class classification problems such as this one.
- Optimizer – this parameter allows us to determine the appropriate learning rate for the neural network. In this case, we have chosen an adaptive learning rate ‘adam’ that computes individual learning rates for different parameters. It is considered a ‘state-of-the-art’ optimization algorithm in recent ML history.

4. TRAINING THE MODEL

The LSTM RNN was trained over 5 epochs. An accuracy of roughly 97% was achieved on the training set over the 5 epochs – 5 epochs were therefore deemed sufficient for initial model training. Below are visualizations of how the loss decreases and accuracy increases (on the training set) over the 5 epochs:





A batch size of 256 was used, which is a the default number to use for such models. This number can be varied/'tuned' to find the value that maximises model accuracy.

Another parameter specified during the training process is the class_weight parameter. This allows us to address any class imbalance in the data by assigning larger weights to those classes with fewer observations in the data. Please see section 5 below for further details on how this parameter was set.

5. TESTING THE MODEL

For testing the model, we make use of Sklearn's train_test_split library that allows us to split the data into a training and testing set. The model is then trained on the training set, while we check its performance on the test set. Seeing that we have relatively few records, we keep 80% (2208 rows) of the data for training and 20% (553 rows) for testing. This is done to give the model a chance to learn from a sufficiently large training set. If we show the model too little data, it might underfit and perform poorly on the test set.

Initially, the class_weight is set to 'auto', where the class weights are automatically assigned. The model is trained and its accuracy calculated on the test set, achieving an **accuracy** (measure of choice) of 97.29% on the test set. We then use Sklearn's class_weight library, allowing us to visualize and explicitly assign the weights. This gives us an accuracy of 97.29% as well. We therefore suspect that setting class_weight = 'auto' does the same as explicitly assigning the class weights using Sklearn.

6. RESULTS

As mentioned in section 5 above, our final model achieves an accuracy of 97.29% on the test set. This is a good accuracy. However, we would expect our model to do well on a relatively easy task such as discriminating between 3 languages.
We also need to consider the class imbalance present in the data. Upon seeing the high accuracy we should still be suspicious of **where** the model performs well/poor. Upon inspecting the accuracy for each language separately, we observe the following:

```
Afrikaans accuracy = 97.69230769230770%
English accuracy = 100.0%
Dutch accuracy = 0%
```

Predicted	Afrikaans	English
Actual		
Afrikaans	127	3
English	0	411
Nederlands	8	4

This is to be expected. The model does very well for English phrases, for which it has ample records to learn from. It still does well for the Afrikaans phrases, which also has some data to learn from. However, the model does not fare well with Dutch phrases – it gets **none** of them right and is unlikely to do so in the future when it is fed previously unseen Dutch phrases. This means that, although the class weights parameter is specified during the model training, additional efforts are required to address the class imbalance (i.e. get the model to predict Dutch phrases with more accuracy). Alternative ways of doing so are discussed in section 7.4 below.

It's worth noting that we could make use of k-fold cross validation to test (and train) our model. This, however, is more time-consuming and requires more computing power.

Finally, we could possibly improve our model's overall performance through hyperparameter tuning. We could make use of Sklearn's GridSearchCV library to help us do so. This process requires specifying a range of values for each parameter. The range of values is then iterated over with the model performance being checked each iteration. The value for that parameter that maximises model performance is made available to the modeler. Hyperparameters that can possibly be 'tuned' is listed below:

- Embedding vector length
- Maximum sentence length
- Memory units in the LSTM layers
- Dropout rate
- Recurrent dropout rate
- Number of layers in the neural network
- Number of epochs
- Training batch size
- Class weights

7. BONUS QUESTIONS

7.1. ALTERNATIVE ML APPROACHES

1. Fit a Support Vector Machine (SVM) to the data

SVMs are often powerful and can scale to larger data sets, but they are also quite complicated and it can be difficult to know what patterns they rely on. This step still relies on the corpus being pre-processed in the same way as described in section 2 above. We would also need to split the data into a train and test set, as was done in our LSTM RNN approach. The model will be tested against the test set using the accuracy measure.

We could also fit a Random Forest to the data since such models often perform relatively well on imbalanced datasets.

2. Automated Machine Learning (AutoML)

We could make use of Google Cloud Natural Language API. This enables us to build (and deploy) custom machine learning models that can analyse documents and classify content. What makes this approach an attractive one is that the user only needs to provide the dataset – the system then takes on the task of identifying the best algorithm (not necessarily a Deep Learning method) and also finding the best hyperparameters.

With Google Cloud AutoML the steps are as follows:

1. The texts are loaded into Google Cloud Storage buckets
2. Using the java-ml-text-utils tool we can export the corpus in CSV format to be interpreted by Google Cloud AutoML
3. Be sure to indicate the labels of our dataset to Google
4. Training starts – Google Cloud AutoML automatically generates a train, test and validation set
5. Hyperparameter tuning the model occurs on the validation set
6. Check the model performance against the test set using accuracy and a confusion matrix

7.2. SUPERVISED VS UNSUPERVISED LEARNING

Supervised learning is where all data is labelled and the algorithms learn to predict the output from the input data. An answer key is provided that the algorithm can use to evaluate its performance on the data. Common types of supervised learning are classification and regression.

Unsupervised learning is where the data is unlabelled and the algorithms learn to inherent structure from the input data on its own. Clustering is a common application of unsupervised learning.

7.3. CLASSIFICATION VS REGRESSION

Classification is the task of predicting a discrete class label. Regression is the task of predicting a continuous quantity. There may be some overlap, for example:

- Classification model that predicts a probability (continuous number) of the input belonging to each of the discrete classes in the data
- Regression model that predicts a discrete value, e.g. an integer

Classification models are evaluated using measures such as accuracy, precision, recall or an F1-score. Regression models are evaluated using measures such as root mean square error and mean absolute error.

7.4. CLASS IMBALANCE: CONSEQUENCES AND SOLUTIONS

Class imbalance poses a challenge for predictive modelling as most of the machine learning algorithms used for classification were designed around the assumption of an equal number of examples for each class. This results in models that have poor predictive performance, specifically for the minority class. As we have seen in our example here, the model has poor predictive performance for the minority of Dutch phrases. In fact, our model is unable to detect **any** of the 12 Dutch phrases in the test set.

It is often the case that the minority class is more important and therefore the problem is more sensitive to classification errors for the minority class than the majority class. For example, identifying fraudulent banking transactions or predicting medical conditions in patients (such as cancer).

Potential methods of addressing our class imbalance include:

1. Collecting more data – a larger dataset can possibly be more balanced, i.e. contain more Dutch phrases for the model to learn from.
2. Resample our dataset – there are two approaches to do so. First, we might add copies of phrases from the Dutch class. We could also delete instances from the over-represented classes, possibly only deleting English records in our example.
3. Generate synthetic samples from instances in the minority (Dutch) class. Systematic algorithms for generating synthetic samples are available, for example the Synthetic Minority Over-sampling Technique (SMOTE)
4. We could try using a different algorithm. Decision trees are believed to perform relatively well on imbalanced datasets.
5. Penalized methods – this is the approach we use in our model. This is where an additional cost is imposed on the model for making classification mistakes on the minority class and can is achieved by setting the ‘class_weight’ parameter when training the model.

7.5. DEEP LEARNING VS NON-DEEP LEARNING APPROACHES

Deep Learning is regarded as part of a broader family of ML and is based on Artificial Neural Networks (ANNs). It often involves many layers of neurons within the ANN, giving the impression of a ‘deep’ network of neurons. This is

where the term ‘Deep Learning’ arises from. Non-Deep Learning methods are considered a subset of classical ML methods and is not characterised by the stacking of many layers of neurons within the algorithm.

Some differences between these two approaches include:

- Deep Learning can be applied to both structured and unstructured data, whereas non-Deep Learning relies on structured data
- Deep Learning scales more effectively with data than non-Deep Learning methods, i.e. model performance takes longer to plateau as the volume of data increases
- Deep Learning methods often have less need for feature engineering than non-Deep Learning methods
- Deep Learning methods make fewer assumptions than non-Deep Learning methods
- Deep Learning methods are usually more difficult to interpret
- Deep Learning methods are usually more computationally expensive