Information Processing and Retrieval

Course Project - Group 25

Frederico Santos
Instituto Superior Técnico
81996

Luís Mateus
Instituto Superior Técnico
82046

Eduardo Cidres
Instituto Superior Técnico
84712

ABSTRACT

This report provides an explanation of the approach used to implement an Information Search & Extraction System for the analysis of political discourse.

1 INTRODUCTION

The goal of the project is to implement in *Python* an Information Search & Extraction System of political discourse. The system receives a large set of documents containing the electoral manifestos of several political parties from different countries in the world and must provide an *Ad hoc* search on the collection of those documents, classify them according to their political affiliation and an analysis of the subjects mentioned.

2 *AD HOC* SEARCH

We implemented the search function on the collection of documents through the use of the Whoosh library. In doing this, we specified the schema to use for storing the manifestos in the index. We also opted to group the contents of manifestos that have the same id in the *csv* file in order to make the index search more clear to the user. To do this, we stored the *csv* fields in dictionaries and concatenating contents where the id was the same, in order to have all the information structured in the way we desired before submitting it to the writing of the index.

When it comes to the search itself we decided to do it in a separate file, because we only need to create the index once in order to search over it.

We chose to keep the weighting model as the default(BM25), as it generally offers better performance than the alternatives given in the API, as well as being more balanced by relying on both term frequency and document length.

By executing the searchCollection.py script we assume that the set of keywords to be searched are the ones typed followed by a space after evoking 'python searchCollection.py', where each keyword is also separated by a space.

We used the default meaning of the Whoosh library for the space between keywords (OR). We also assume that the keywords are typed correctly, and the only processing that is made from the input is the conversion of keywords to lower case in order to facilitate the search.

In order to get the manifesto count per party, we chose to use the Whoosh embedded option in the search function, where it allows us to obtain a grouping of the search results of the specified schema field in the function call.

For the keyword count per party, we ended up implementing it fully ourselves since we thought none of the options provided by Whoosh were as viable as making our own code. To do this we made use of dictionaries, storing counts of the keywords mentioned in party's manifestos read from the Results object.

3 CLASSIFICATION

For the classification process we split the work in two functionalities: **predict which political party** is most likely to have produced a given natural language text and **test the effectiveness of the predictions** by showing precision, recall and F1 values.

3.1 Political Party Prediction

For this first part we expect to receive a natural language text and, with it find the most likely party to have produced it. To achieve a good performance, we start by preprocessing, tokenizing and filtering stop-words from all the manifesto's text using the CountVectorizer from the sklearn library that builds a dictionary of features and transforms the text to feature vectors.

Sequentially, we apply the TFIDF (Term Frequency times Inverse Document Frequency) downscaling to reduce the weight of the words occurrence count values in larger documents (*i.e.,* we divide the number of occurrences of each word in a document by the sum of words in the document) and also reduce the weight of the words that occur in many documents and therefore are less relevant, using the TfidfTransformer from the sklearn library.

Finally, we train a chosen classifier (explained later in Results) to try to predict the most likely party to have written the input text. To achieve better accuracy we format the input text by removing the stop-words and punctuation before giving it to the classifier. The more specific and informative the input text is, the most accurate will be the prediction.

3.2 Effectiveness of Predictions

In this functionality we use the same process as the one described in the previous sub-section (*CountVectorizer -> TFIDFTranformer -> TrainClassifier*) with the difference that we split the data from the *csv* in a train set (80%) and in a test set (20%). The classifier (trained with the train set) then receives the texts from the test set and returns a prediction array that is later compared with the correct array to evaluate the precision, recall and F1 scores.

3.3 Run the Script

To run the Classification script run the command 'python classification.py <text>' where text is the natural language text given to predict the party.
If no text is given the script will run the Effectiveness of Predictions part and return the precision, F1 and recall values as well as a confusion matrix (explained in Results section). If a text is given the script will return the previously discussed values and the most likely party to have produced the input text.

4 ENTITY ANALYSIS

The entity recognition process works in two stages. The first one uses the ne_chunk function from the NLTK library. What is done here is chunking and processing the tagged tokens from the manifestos (through the use of pos_tag and word_tokenize functions, respectively).

The output of this sequence is stored in an object with the following structure:
[NameOfParty:[(NamedEntity, CallCount)]]

This allows us to easily iterate through the entities without losing track of whom they belong to, whilst also tracking how many times they are being written indiscriminately through all the manifestos. This answers the purpose of the last question.

On the second stage, we simply process this object through a series of sortings to find the results we want, such as the most mentioned entity through all the parties.

Performance-wise, there were several attempts at optimizing it, but they fell short since the main processing culprit is the ne_chunk function, which is at the core of the solution we want to implement. We made an attempt at implementing a solution involving the spaCy library, but we encountered a bug in which spaCy consumes an overly excessive amount of memory, making it inviable for our solution.

5 RESULTS

5.1 Search Solution

Examples:
Success-> python searchCollection.py uk russia war
Success-> python searchCollection.py UK ruSsIa waR
Unsuccessful-> python searchCollection.py uk rassia wer
Unsuccessful-> python searchCollection.py ukrussia war
Unsuccessful-> python searchCollection.py where the as which

Since we assume the input has no typos or stop words, if these do appear in the input, the results are either nonexistent or with very little relevance(stop words)

5.2 Classification Solution

For the classifiers we tested the Stochastic Gradient Descent (SGD) model, the Linear Support Vector Classification (LinearSVC) and the Naive Bayes classifier for multinomial models (MultinomialNB) where we reached accuracy levels of approximately 51%, 52% and 42% respectively, consequently we found the LinearSVC to be the best fitting model, due to best accuracy and average precision, recall and f1 scores **(table 1)**.

Party	Precision	Recall	F1-Score	Support
Conservative Party	0.49	0.44	0.47	385
Democratic Unionist Party	0.46	0.26	0.33	47
Green Party of England and Wales	0.53	0.57	0.55	478
Labour Party	0.49	0.55	0.51	555
Liberal Democrats	0.41	0.50	0.45	569
Scottish National Party	0.63	0.62	0.62	554
Social Democratic and Labour Party	0.70	0.33	0.45	94
The Party of Wales	0.69	0.52	0.59	187
Ulster Unionist Party	0.57	0.25	0.34	106
United Kingdom Independence Party	0.52	0.49	0.50	315
We Ourselves	0.54	0.55	0.55	56
avg / total	0.53	0.52	0.52	3346

Table 1: Classification Report

Each row of the confusion matrix **(Matrix 1)** represents the instances in a predicted party while each column represents the instances in an actual party (*i.e.,* the Correct Predictions are placed in the diagonal, a row identifies the False Positives and a column identifies the False Negatives per party).
We can see from the resulting matrix that the diagonal has the majority of the predicted values, meaning that we have a good amount of correct predictions.

Unfortunately, due to time constraints we were not able to compare the matrix generated to the ones from other classifiers or to test different parameters on each classifier.

$$\begin{bmatrix} 170 & 2 & 23 & 60 & 83 & 20 & 0 & 4 & 1 & 20 & 2 \\ 8 & 12 & 5 & 7 & 7 & 6 & 0 & 0 & 0 & 0 & 2 \\ 19 & 0 & 273 & 47 & 67 & 38 & 2 & 7 & 2 & 21 & 2 \\ 34 & 3 & 35 & 303 & 109 & 34 & 3 & 9 & 2 & 22 & 1 \\ 38 & 2 & 68 & 82 & 287 & 39 & 0 & 14 & 4 & 31 & 4 \\ 22 & 2 & 36 & 57 & 51 & 344 & 4 & 6 & 3 & 24 & 5 \\ 6 & 2 & 6 & 10 & 11 & 18 & 31 & 1 & 3 & 3 & 3 \\ 8 & 0 & 12 & 13 & 37 & 9 & 1 & 98 & 2 & 6 & 1 \\ 11 & 2 & 16 & 12 & 12 & 9 & 2 & 2 & 26 & 10 & 4 \\ 28 & 0 & 33 & 29 & 35 & 29 & 1 & 2 & 3 & 153 & 2 \\ 2 & 1 & 4 & 4 & 9 & 3 & 0 & 0 & 0 & 2 & 31 \end{bmatrix}$$

Matrix 1: Confusion Matrix

5.3 Entity Analysis Solution

By running the statisticalAnalysis.py, we can see the outcome of 4 different statistics that show some insight at how the manifestos are written. The library itself, although not shown purely by the output of the statistical functions, internally has a hard time dealing with certain things. Among these are, for example, fully capitalized words. Another interesting observation is that the manifestos themselves are written in a way that is hard to make a proper evaluation of the statistics, if we use a blind approach like we did. An example

of this, is on the output of the most mentioned entities per party. The most mentioned entity by the United Kingdom Independence Party is UKIP, but the system will not recognize UKIP as being the party itself. No inputs are required to run this process.