# Task 3 of the 2009 edition of the Text Mining Challenge (DEFT): learning classification by political party of interventions in the European Parliament

**Asil QRAINI**
M2 NLP, Paris Nanterre University
`asilqraini@gmail.com`

## Abstract

This project was carried out as an implementation of the knowledge obtained during the class "Apprentissage Artificiel" with Loïc Grobol for the first semester of the second year of masters in the NLP program at Paris Nanterre University.

***Keywords*—** Opinion mining, text classification, machine learning

## 1 Introduction

With all the technological advancements we're having in the past few years, resources and amounts of data substantially increased, leaving the task of understanding and analyzing it difficult and time-consuming because of the unstructured nature of said data. In order to overcome this hurdle, we can deploy automatic technique such as Text Classification. Text Classification is a machine learning technique for assigning predefined categories or labels to unstructured text data. It is a subfield of natural language processing (NLP) and information retrieval (IR).

The goal of the third task in the DEFT challenge is to automatically classify interventions made by members of the European Parliament (MEPs) into different political parties based on the text of the interventions.

## 2 Procedure

In order to proceed in this task, we needed to use the dataset provided, which contained debates made by MEPs along with their labels (each intervention with the political party of the MEP who made it). We then used this labeled dataset to train a machine learning model to classify interventions as belonging to a particular political party.

Once the model was trained, we used it to classify new, unseen interventions made by MEPs and predicted the political party of the MEP based on the content of the intervention. This task was approached using supervised learning, where the labeled dataset of MEP interventions is used to train a classifier to predict the political party of a given intervention.

### 2.1 Data

The data used consists of a corpus of debates which was constituted by recovering, for each parliamentary session of the period from 1999 to 2004 the agenda, from which the transcript of the debates were recovered, in the three languages foreseen for the challenge (French, English and Italian). We had this part already done by the organizers of the challenge and only had to use the data. The data was already separated into three different files:

1. Training corpus: containing tags for the political parties, the IDs, as well as the texts in an XML format.
2. Test corpus: containing tags for the texts and the IDs, also in an XML format.
3. Reference corpus: a text file containing all the IDs of the political parties that reference to those of the test corpus.

### 2.2 Platform

In order to facilitate the work as a group, we chose to use Google Colab as more than one person can work simultaneously on the notebook as well as see the changes/modifications of others in real-time.

### 2.3 Language choice

We chose to work on the English corpus out of the three available languages as we're more familiar with it, given the fact the both team members are not native French speakers. In addition, we believed that working on English as start would be easier and that in the future, we could modify the algorithm to also include Italian and French.

### 2.4 Libraries used

- **"spacy_sentence_bert"**: We opted to use a spaCy package, which is a free, open-source library for natural language processing (NLP) in Python. It provides tools for tokenization, part-of-speech tagging, dependency parsing, and more. The ***"spacy_sentence_bert"*** package is a spaCy extension that allows users to use the BERT model, a pre-trained transformer model developed by Google, for various NLP tasks such as text classification, entity recognition, and question answering.

- **BeautifulSoup**: since both the train and test corpora are in an XML format, we used the BeautifulSoup library to read them. This library is usually used for web scraping and parsing HTML and XML documents. It is designed to make it easier to navigate, search, and modify the parse tree of an HTML or XML document.

- **NLTK**: this library provides tools for tokenizing, parsing, and tagging text, as well as for implementing and evaluating models for text classification, language modeling, and other natural language processing tasks. In our case, we've used it for the tokenization and cleaning part (removal of stop words) by installing specific packages (stopwords and punkt).

- **NumPy and pandas**: those libraries are often used together in data analysis and machine learning workflows, as they provide a powerful and flexible set of tools for manipulating and analyzing numerical data.

- **Scikit-learn**: provides a range of tools and techniques for implementing and evaluating machine learning models, including classification, regression, clustering, and dimensionality reduction. We've used the following functions and models from this library:

1. **"sklearn.metrics.accuracy_score"**: we used this function to evaluate the performance of our classification model.It calculated the fraction of predictions that are correct, also known as the accuracy.
2. **"TfidfVectorizer"**: we've used "TfidfVectorizer", which is a transformer that converts a collection of raw documents into a matrix of TF-IDF (term frequency-inverse document frequency) features.
3. **"sklearn.svm"**: we imported the Support Vector Classification (SVC) model, which is a linear classifier that is trained using the "one-vs-one" approach for multi-class classification.
4. **"LogisticRegression"**: we imported the Logistic regression algorithm that is usually used to predict the probability of a binary outcome, such as the likelihood that a customer will churn or that a patient will have a disease.
5. **"Random Fores"**: we imported the Random Forest algorithm that is used for classification and regression tasks. This algorithm combines the predictions of multiple individual models to make a final prediction.
6. **"Naïve Bayes"**: we also imported the Naive Bayes algorithm. It is based on the Bayes theorem, which states that the probability of an event (in this case, the class label of a data point) is equal to the prior probability of the event multiplied by the likelihood of the event given the data.

- **"TQDM"**: we used this with pandas to visualize the progress of a loop or iteration that processed a pandas DataFrame. It displayed a graphical representation of the progress, along with an estimated time remaining, and allowed us to see how far along the loop or iteration is.

## 3 Application

### 3.1 Setting the notebook and reading the data

Since we've used **Google Colab**, the first step was to mount the drive and then import the data. We're also installed all necessary libraries mentioned in section **2.4**. We then sat the seed for the NumPy random number generator at 500. By doing that, we ensured that the generator produces the same sequence of random numbers every time the code is run. We then set the language to English, since the texts we're treating are only in said language. This could be modified to treat other languages. We then read the dataset from the relevant **XML** files and parsed the training data and test data using BeautifulSoup.

### 3.2 Finding the data needed

Since the **XML** data files contained elements and tags that we're necessary for our task, we had to find all relevant information from the **<parti>** and **<texte>** tags by using the ***"findAll()"*** method of the Beautiful Soup library to search for all elements in both tags.

### 3.3 Cleaning the data

Cleaning data before classification is important because it can help to improve the accuracy and effectiveness of the classification model. It can also help to reduce the amount of noise or irrelevant information in the data, which can improve the model's ability to generalize to new data. This is particularly important if the classification model is being trained on a small dataset, such as ours, as the model may be more sensitive to noise or irrelevant information in the data. We first converted all the characters in the string to lowercase, then applied a regex catchall in order to remove all unnecessary punctuation. We specified the punctuation to be removed in a variable as it was not recognizing some of them, such as (**"** and **'**).

### 3.4 Cleaning the labels

We read the ***"train_ids"*** file into a pandas DataFrame and stored it in the ***"y_train_labels"*** variable. We then added column headings (Id and Party). After that, we turned those DataFrames into arrays. We then built the test datasets by extracting the ***"valeur"***, which are the names of the political parties, and stored them in the ***"y_test"*** variable.

### 3.5 Exploratory Data Analysis

We analyzed and summarized our dataset in order to understand the distribution, patterns, and relationships present in the data. But before doing that, we had to store our train and test data in separate pandas DataFrames. We created a bar plot of the counts of the values in the ***"y_train"*** and the ***"y_test"*** DataFrames. The graphs demonstrated that we have a majority of ***"PPE-DE"*** and ***"PSE"*** which suggests that our models may classify the other three poorly depending on the verbosity of the features, therefore we'll shuffle the dataset.

### 3.6 Shuffling the dataset

We used the ***"sample()"*** method, which is a pandas method that randomly samples rows from the DataFrame, and the ***"frac"*** argument specifies the fraction of rows to include in the sample. In this case, "frac=1" specifies that all rows should be included in the sample, so the effect of this line of code is to shuffle the rows of the DataFrame. After that, we applied the ***"dropna()"*** method in order to remove rows with missing values from the DataFrame. We then used the ***"word_tokenize()"*** function from the Natural Language Toolkit **(NLTK)** library that tokenized the strings into lists of words. The last step before moving on to the vectorization was to clean our data from stopwords. We used the ***"progress_apply()"*** that applies a function to each element in a column and displays a progress bar, and then the ***"remove_stop_words()"*** function that removes stop words from a list of tokens.

### 3.7 Vectorizing with transformer

We started by loading the ***"en_stsb_distilbert_base"*** model, which is a BERT spaCy sentence transformer. After loading the transformer, we vectorized our sentences. We first converted our list of strings to strings using the ***"apply(list_to_string)"*** pandas function. We vectorized usin the ***"vectorize()"*** function, which uses the ***"nlp()"*** function from the ***"spacy_sentence_bert"*** library to process a string and return its vector representation. We also used the ***"progress_apply()"*** which we've applied earlier.

### 3.8 Creating the classifier

The Support Vector Machine (SVM) classifier from ***"sklearn.svm"*** was built using the following code:

```
clf = SVC(gamma='auto', verbose=True)
clf.fit(train_df["x_train_vector"].to_list(),train_df["y_train"].to_list())
y_pred = clf.predict(test_df["x_test_vector"].to_list())
```

The ***"gamma="auto""*** argument specifies that the kernel coefficient for the SVM should be chosen automatically, and the ***"verbose=True"*** argument specifies that the classifier should print progress messages during training. The ***"accuracy_score()"*** function from the ***"sklearn.metrics"*** module that takes the true labels and predicted labels for a dataset as input and returns the accuracy of the predictions. The ***"np.round()"*** function is a function from the ***numpy*** module that rounds a number to the specified number of decimal places. The ***"*100 and decimals=4"***arguments specify that the accuracy should be expressed as a percentage with **4** decimal places.

## 4 Results

After having using the **SVM** classifier, we got an accuracy score of **40.9138%**, which is considered low. One of the reasons as to why this may have happened is insufficient training data. We may not have had enough training data, and so the classifier was not able to learn the relationships between the input features and the labels accurately. Another reason could be that the **SVM** classifier is not a good fit for the data, which is why we applied three other classifiers and got the following results:

1. **Random Forest**: 63.3764% after the initial 49.0294%. The increase of the accuracy was due to the increase of the "***max_depth***" from **9** to **none**, which means that the trees in the random forest can grow until all the leaves are pure, or until all the leaves contain the minimum number of samples required to split a node.

2. **Logistic Regression**: 44.9768%. We are not sure if this low score was a result of the model may be overfitting to the training data, which means that it is performing well on the training data but not generalizing well to new, unseen data. This could be due to having too many features, or a lack of regularization. Or if the model is underfitting to the data, which means that it is not able to capture the underlying patterns in the data. This could be due to having too few features, or a lack of complexity in the model.

3. **Naïve Bayes**: 31.6624%. This very low score could be a result of the fact that the assumption of independence between features was not holding for the data being used. Naive Bayes assumes that the features are independent, which means that the presence or absence of one feature does not depend on the presence or absence of any other feature. If this assumption is not true, the model's predictions may be less accurate. Otherwise, it could be that the model may be oversimplified for the problem at hand. Naive Bayes is a simple and fast model, but it may not be able to capture the complexity of the data if the relationships between features are more complex.

We can tell from the results that the Random Forest model had the highest accuracy score. It could be because of the fact that in a Random Forest model, the individual models are decision trees, which are trained on different subsets of the training data and with different subsets of the features. In addition, this model is usually able to capture the complexity of the complex relationships between features and the target variable, which can lead to better performance compared to simpler models. Random Forest can also can handle imbalanced classes, which may be an issue for other models. Lastly, Random forests are able to handle missing or incomplete data by constructing decision trees based on the available features.

## Limitations

Although we've only worked on English, this method should easily work on other languages such as French and Italian. It's important to mention that this kind of classification requires large **GPU** resources, which we do not have access to at the moment. The before mentioned caused slower training, out-of-memory errors and limited model complexity. There are several possible ways to improve the performance of our classifiers, such as:

- Use more data: this can help the model learn more about the underlying patterns in the data, and can lead to better generalization to new, unseen data.

- Use hyperparameter tuning: it involves adjusting the parameters of the model to find the best combination of hyperparameters. This can be done using techniques such as grid search or random search.

- Use regularization: regularization is a method of preventing overfitting by adding a penalty term to the objective function. This can help the model generalize better to new data.

- Use different algorithms: it may be worth trying different algorithms to see if you can find one that performs better.