Ximena Moure Alassio
Maria Paraskeva

Language Detection

Introduction

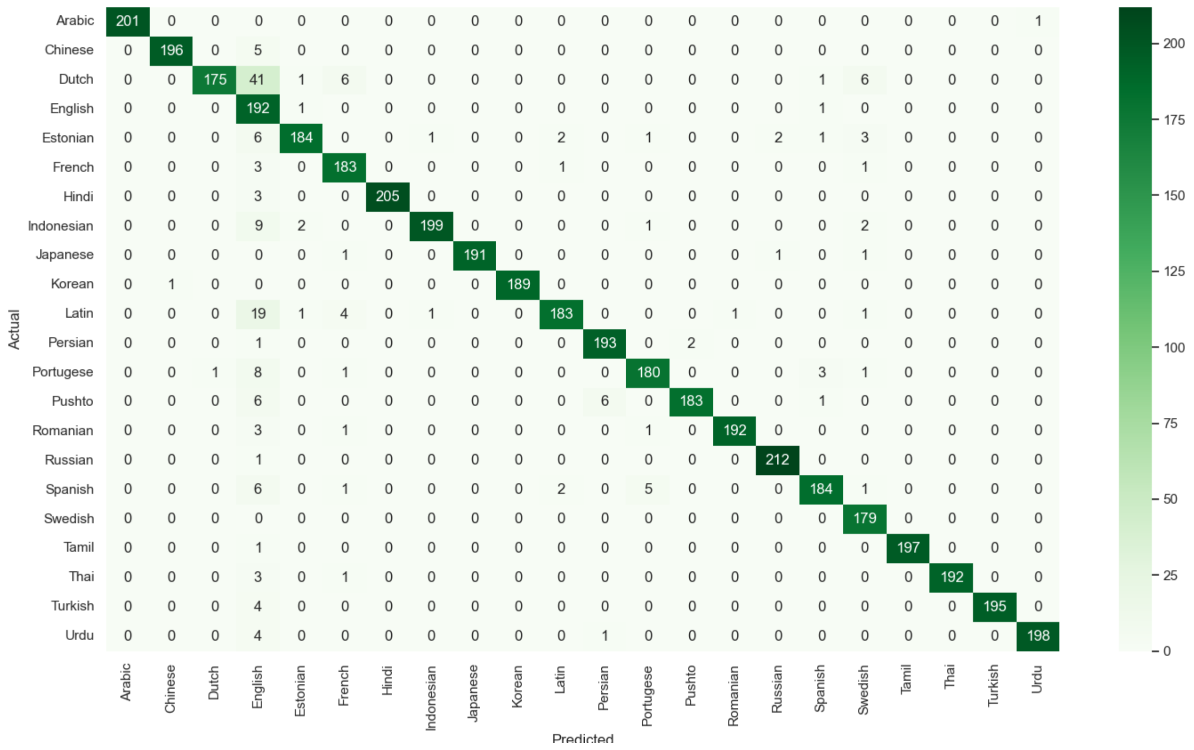
The main objective of this report is to get a better understanding and get familiar with Natural Language Processing (NPL) techniques and tools.
In order to achieve this, we first start analyzing an already working system by playing around with the parameters (vocabulary size and unigram granularity). Then, as a second step we apply different preprocessing techniques. Lastly, we implement alternative classifiers to Naive Bayes. This is all done with the goal of analyzing the different effects they have.

First Baseline

Character granularity

In this section we analyze the language detection based on characters. We start with the base case which is for 1000 characters and then we modify the number of characters to be 10000 and 100, and compare the results.

Looking at the results when analyzing a unigram with length 1000, we can see that it has a coverage of 98%, which means that almost all characters in the corpus are featured in the vocabulary. Another thing we can notice is that it has a weighted F1-score of 0.95741 which is very good.



Actual \ Predicted	Arabic	Chinese	Dutch	English	Estonian	French	Hindi	Indonesian	Japanese	Korean	Latin	Persian	Portugese	Pushto	Romanian	Russian	Spanish	Swedish	Tamil	Thai	Turkish	Urdu
Arabic	201	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
Chinese	0	196	0	5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Dutch	0	0	175	41	1	6	0	0	0	0	0	0	0	0	0	0	1	6	0	0	0	0
English	0	0	0	192	1	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
Estonian	0	0	0	6	184	0	0	1	0	0	2	0	1	0	0	2	1	3	0	0	0	0
French	0	0	0	3	0	183	0	0	0	0	1	0	0	0	0	0	0	1	0	0	0	0
Hindi	0	0	0	3	0	0	205	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Indonesian	0	0	0	9	2	0	0	199	0	0	0	0	1	0	0	0	0	2	0	0	0	0
Japanese	0	0	0	0	0	1	0	0	191	0	0	0	0	0	0	1	0	1	0	0	0	0
Korean	0	1	0	0	0	0	0	0	0	189	0	0	0	0	0	0	0	0	0	0	0	0
Latin	0	0	0	19	1	4	0	1	0	0	183	0	0	0	1	0	0	1	0	0	0	0
Persian	0	0	0	1	0	0	0	0	0	0	0	193	0	2	0	0	0	0	0	0	0	0
Portugese	0	0	1	8	0	1	0	0	0	0	0	0	180	0	0	0	3	1	0	0	0	0
Pushto	0	0	0	6	0	0	0	0	0	0	0	6	0	183	0	0	1	0	0	0	0	0
Romanian	0	0	0	3	0	1	0	0	0	0	0	0	1	0	192	0	0	0	0	0	0	0
Russian	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	212	0	0	0	0	0	0
Spanish	0	0	0	6	0	1	0	0	0	0	2	0	5	0	0	0	184	1	0	0	0	0
Swedish	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	179	0	0	0	0
Tamil	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	197	0	0	0
Thai	0	0	0	3	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	192	0	0
Turkish	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	195	0
Urdu	0	0	0	4	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	198

Figure 1: Confusion matrix of the baseline classification with 1000 characters

The confusion matrix (figure 1) shows that the algorithm performs quite well. For most of the languages it classifies it correctly. However, we can observe that the most noticeable misclassification is that it classifies Dutch as English. Doing some research, we read that both English and Dutch belong to the Germanic branch of the Indo-European language family, this is why they share some common linguistic features. Additionally, many words in English have roots in Deutch, such as “cookie” (koekje) and “boss” (baas). As a result, there are many words that have the same origin and meaning in both languages.
It can also be seen that there is another misclassification between English and Latin. This may be due to the fact that English has many words that have roots in Latin. Another reason may be due to errors on the dataset. When we took a closer look at the data, we noticed that some sentences containing English text are labeled as Latin and that some sentences that are not in English contain sub-sentences or English words.

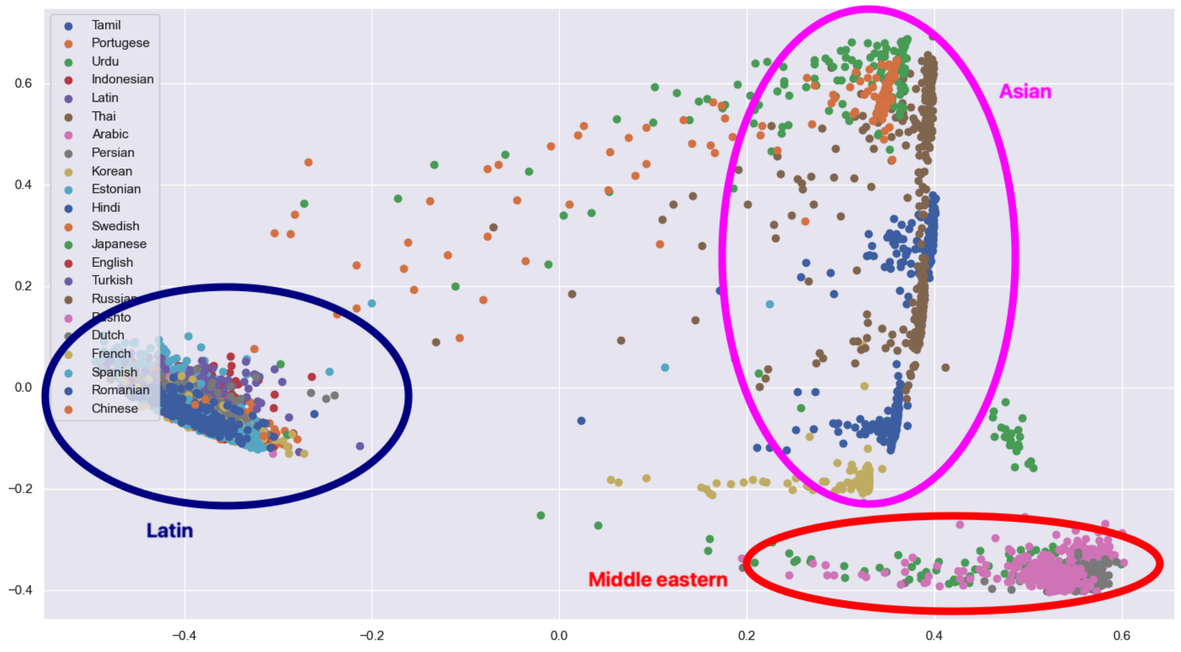


Figure 2: PCA of the baseline classification using 1000 characters.

From the Principal Component Analysis (PCA) results and the plot (figure 2) we can observe that two dimensions cover approximately 45% of the variance in the training set. The first dimension, which explains 31.3% of the variance, can be interpreted as the difference between the languages that use the Latin alphabet and the ones that don´t. In the second dimension, which explains 13.8% of the variance, we can see that when it only considers Asian languages it can distinguish, to some degree, the difference between them (the points of the right top cluster in figure 2 are mostly well separated).

We performed a second analysis which consisted in increasing the character vocabulary size to 10000. When doing so from the output we can see that only 6818, which are the total characters, are included.
The weighted F1-score is 0.9202, which is worse than the one we obtained with 1000 characters. In the confusion matrix, we can see that now Dutch gets misclassified as English more times than before and that it also confuses Dutch with Swedish. Considering the results, we can conclude that adding more unigrams doesn't seem to have a positive impact on the algorithm. As we add more unigrams, the influence on the probability of a language decreases, so unigrams that were influential before are now surpassed by the additional ones.

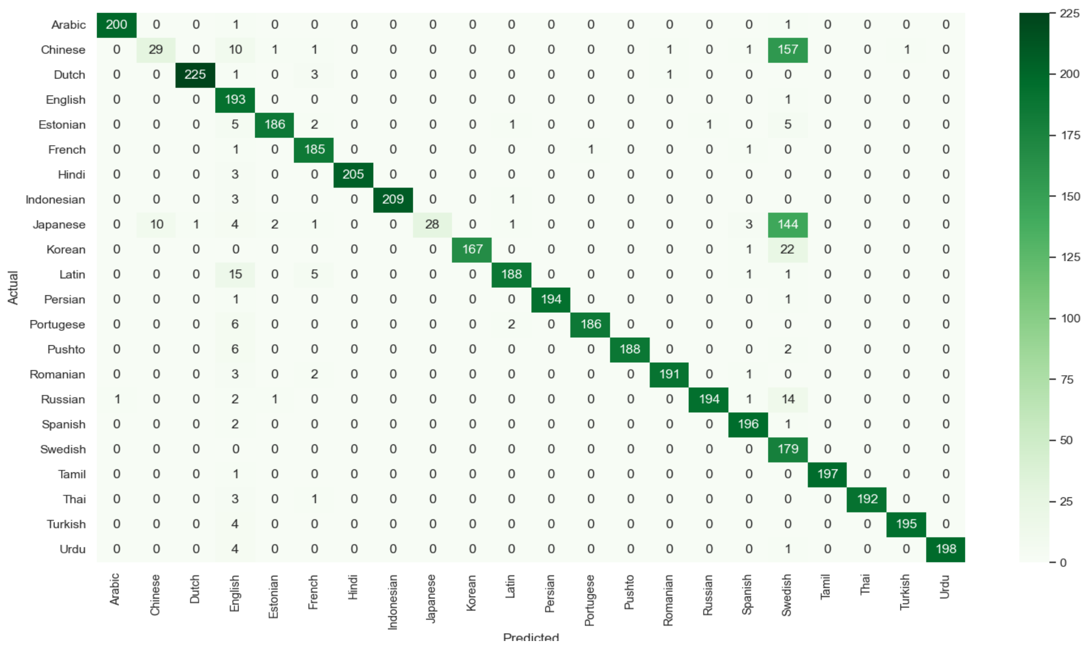
Lastly, we decreased the character vocabulary size to 100. The weighted F1-score now decreased to 0.8331 which is worse than both previous tests. Taking a look at the confusion matrix we see that Dutch misclassification has decreased a lot but there are new issues with Chinese and Japanese getting misclassified as Swedish. Even when having an 82% coverage, there are many small misclassifications with at least half of the languages, so the conclusion is that reducing the character size does not lead to better results.

Word granularity

In this section we analyze language detection based on words. We start with the base case which is for 1000 words and then we modify the number of words to 20000 and compare the results.

We first notice that now the coverage is reduced to 25.77% and that the weighted F1-score is also reduced to 0.88457. So based on this, we can expect that the results will be worse than when the unigram vector contained characters.

The confusion matrix (figure 3) shows that the most noticeable misclassifications are in Chinese, Japanese and Korean, since the algorithm labeled numerous of them as Swedish. This may be due to the fact that these languages have a character-based writing system which leads to the algorithm tokenizing sentences in these languages as words. The probability of having repeated sentences is small, so when provided with character-based languages, the sentences do not contain words that are in the vocabulary and therefore the classifier has to essentially make a guess. As they are classified as Swedish, we can say that the algorithm seems to prefer Swedish when guessing. If we change the seed, to for example 200, and we run the analysis again, the preferred language to make the guess is now Indonesian.



	Arabic	Chinese	Dutch	English	Estonian	French	Hindi	Indonesian	Japanese	Korean	Latin	Persian	Portugese	Pushto	Romanian	Russian	Spanish	Swedish	Tamil	Thai	Turkish	Urdu
Arabic	200	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Chinese	0	29	0	10	1	1	0	0	0	0	0	0	0	0	1	0	1	157	0	0	1	0
Dutch	0	0	225	1	0	3	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
English	0	0	0	193	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Estonian	0	0	0	5	186	2	0	0	0	0	1	0	0	0	0	1	0	5	0	0	0	0
French	0	0	0	1	0	185	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0	0
Hindi	0	0	0	3	0	0	205	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Indonesian	0	0	0	3	0	0	0	209	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Japanese	0	10	1	4	2	1	0	0	28	0	1	0	0	0	0	0	3	144	0	0	0	0
Korean	0	0	0	0	0	0	0	0	0	167	0	0	0	0	0	0	1	22	0	0	0	0
Latin	0	0	0	15	0	5	0	0	0	0	188	0	0	0	0	0	1	1	0	0	0	0
Persian	0	0	0	1	0	0	0	0	0	0	0	194	0	0	0	0	0	1	0	0	0	0
Portugese	0	0	0	6	0	0	0	0	0	0	2	0	186	0	0	0	0	0	0	0	0	0
Pushto	0	0	0	6	0	0	0	0	0	0	0	0	0	188	0	0	0	2	0	0	0	0
Romanian	0	0	0	3	0	2	0	0	0	0	0	0	0	0	191	0	1	0	0	0	0	0
Russian	1	0	0	2	1	0	0	0	0	0	0	0	0	0	0	194	1	14	0	0	0	0
Spanish	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	196	1	0	0	0	0
Swedish	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	179	0	0	0	0
Tamil	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	197	0	0	0
Thai	0	0	0	3	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	192	0	0
Turkish	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	195	0
Urdu	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	198

Figure 3: Confusion matrix of the baseline classification with 1000 words

The PCA analysis shows that the two dimensions cover approximately 11,5% of the variance. This is also shown in figure 4, where we can see that it is not possible to distinguish different clusters, so we are not able to draw any conclusion from it.

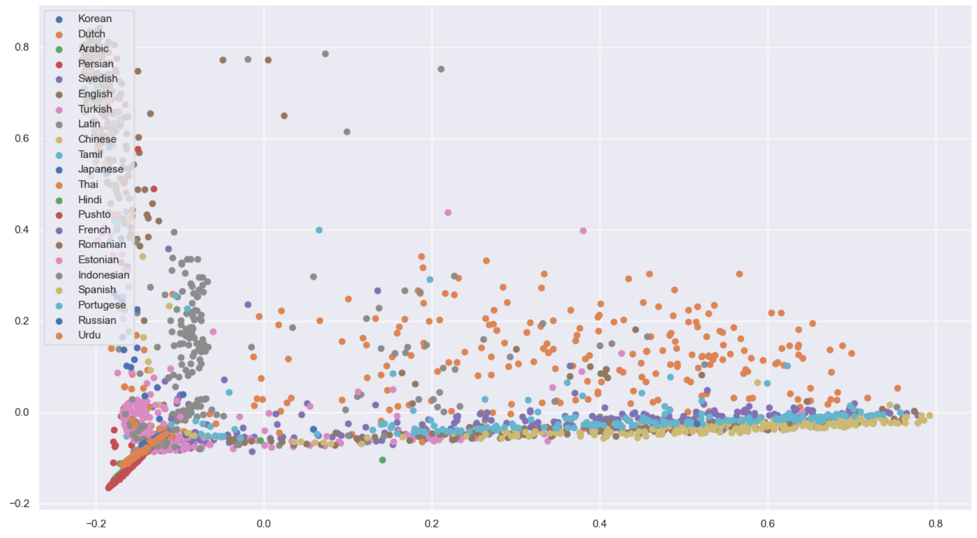


Figure 4: PCA of the baseline classification using 1000 words.

To see how the quantity of words in the vocabulary affects the results we increased it to 20000. As a result, the coverage increased to 49.6% and the weighted F1-score is now 0.91772. This is a modest improvement compared to using 1000 words, but it is still not good enough as when using characters. From the confusion matrix we can see that it stills misclassifies character-based languages, mostly Chinese and Japanese. The PCA plot is quite similar to the one in figure 4 and the explained variance decreased to approximately 7.5%.

We can conclude that when using the Naive Bayes classifier, the character-based language offers better results.

Preprocessing

The different preprocessing strategies that we applied are described in this section. To implement the different strategies, we took advantage of all the functionalities that the library NLTK (Natural Language Toolkit) provides.
All the strategies were applied considering a vocabulary of 1000 words.

Removal of punctuation

We started with a simple technique which was to remove all punctuation, which includes symbols and numbers.

Lowercase

We were going to convert all characters to lowercase but the class ‘CountVectorizer’ from the ‘sklearn’ package already does it by default, so we skipped this.

Remove Stop words

Stop words are frequently used words that carry very little meaning; they are words that are very common in a specific language. They are typically articles, prepositions, pronouns, conjunctions, etc. which are used a lot but do not add lots of information to the analysis.

Stemming

We decided to apply stemming, which is the process of reducing words to its base root form (stem), by removing suffixes, affixes or prefixes that may be attached to it. To do this, we used the Porter Stemmer that works by applying a set of heuristics rules to reduce words to their base root form. Porter Stemmer is relatively fast and simple to implement.
Stemming can help in reducing the number of unique words in the corpus. Words that are essentially the same are treated as a single token.

Lemmatization

Lemmatization reduces a word to its base or dictionary form (lemma) by applying morphological analysis and language rules. The main difference between lemmatization and stemming is that the former produces a valid base word or lemma, while the latter may produce a truncated or inaccurate base form. So, we can say that stemming only looks at the form of the world whereas lemmatization looks at the meaning of the word.

Since the rules for forming the base form of a word can vary depending on the language, to apply lemmatization, the language of the text needs to be known. So, even though we considered applying lemmatization we reached the conclusion that it was not viable.

Sentence tokenization

To carry this step, we used the method 'sent_tokenize' from the nltk library. When doing this, we took into consideration that sentence splitting affects the number of sentences and therefore we must replicate the labels to match.
By applying sentence tokenization, it becomes possible to apply different natural language processing techniques at a more granular level, which can lead to more accurate results.

Results

As a conclusion, we can say that the different preprocessing techniques did not improve the coverage of the F1 score (see table 1). We believe that one possible reason for this can be that the data is already clean.
We also tried combinations of the different techniques, but the results were pretty much the same; the F1 score did not improve.

Additionally, as we mentioned before in the baseline exercise, the most noticeable misclassifications continue to be Chinese, Japanese, and Korean. Maybe we could have applied some techniques to try to improve the tokenization of these languages. In these languages a character represents a word, and a set of characters represent a sentence, so we could look at the alphabet used in the sentence instead of looking at the labels.

	None	Stemming	Sentence tokenization	Stop words	Punctuation
Coverage	0.2577	0.2443	0.2577	0.1434	0.2579
Micro F1 score	0.8921	0.8927	0.8921	0.7586	0.8923
Macro F1 score	0.8812	0.8822	0.8812	0.7643	0.8814
Weighted F1 score	0.8846	0.8856	0.8846	0.7664	0.8847
PCA 1	7.88%	8.05%	7.87%	2.65%	7.88%
PCA 2	3.64%	3.49%	3.63%	2.40%	3.65%

Table 1: Preprocessing results

Classifiers

In this section we describe the different classifiers that were applied to the baseline to the results after applying the different preprocessing strategies.

Naïve Bayes

The classifier provided in the task was the Naïve Bayes classifier, which is a probabilistic classification algorithm based on Bayes theorem. It assumes that the occurrence or absence of a feature does not influence the presence or absence of some other feature.

Linear Support Vector Classifier

This classification algorithm uses a kernel to map the data into a high-dimensional space and then finds a linear hyperplane that separates the classes. We decided to use it because it has a good performance, and it is efficient on high-dimensional and sparse data.

K-Nearest Neighbor

It finds the k nearest points in the training set to the new point and predicts the class of the new point based on the classes of its k nearest neighbors. We used the class 'KNeighborsClassifier' from 'sklearn.neighbors' and we applied it with the default k, which is 5.

Decision Tree

It works by recursively partitioning the input space into smaller regions by selecting the feature and threshold that provide the maximum information gain at each node of the tree.
We used 'DecisionTreeClassifier' from 'sklearn.tree' to implement it.

Random Forest

To implement this we used the 'sklearn.ensemble.RandomForestClassifier'. We decided to leave the number of trees in the forest to the default which is 100.

Neural Network

We applied a neural network algorithm using the MLPClassifier class from 'sklearn.neural_netowrk'. The Multi-Layer Perceptron Classifier consists of one or more layers of artificial neurons, each of which is connected to the neurons in the previous and next layer by weighted connections.

Code

Python
```python
#Preprocess.py

# Get stop words for the languages obtained from the parameter
label
def get_stop_words(labels):
    available_stop_words_nltk = stopwords.fileids()
    different_labels = labels.unique().tolist()
    languages = [lang for lang in different_labels if
lang.lower() in available_stop_words_nltk]
    stop_words = list(set([word.lower() for lang in languages
for word in stopwords.words(lang)]))
    return stop_words

# remove stop words
def remove_stop_words(sentence, labels):
    stop_words = get_stop_words(labels)
    tokens = nltk.word_tokenize(sentence)
    filtered = [w for w in tokens if not w in stop_words]
    filtered = " ".join(filtered)
    return filtered
# Keep in mind that sentence splitting affects the number of
sentences
# and therefore, you should replicate labels to match.
def split_sentences(sentences, labels):
    result_sentences =
list(itertools.chain.from_iterable(nltk.sent_tokenize(text)
for text in sentences))
    result_labels = [label for text_sentences, label in
zip(sentences, labels) for _ in
range(len(nltk.sent_tokenize(text_sentences)))]
    return result_sentences, result_labels

def remove_numbers_symbols(text):
    table = str.maketrans('', '', string.punctuation +
string.digits)
    return text.translate(table)

#Tokenizer function. You can add here different preprocesses.
def preprocess(sentence, labels, stopwords=True,
stemming=True, punkt_token= True, split_sentence= True,
remove_symbols=True ):
    if split_sentence == 'True':
        sentence,labels = split_sentences(sentence, labels)
    texts_results = []
    # remove punctuation
    for text in sentence:
        # we don't need to lower case because CountVectorizer
already does it by default
        if remove_symbols == 'True':
            text = remove_numbers_symbols(text)
        if stopwords == 'True':
            text = remove_stop_words(text, labels)
        if stemming == 'True':
            stemmer = nltk.stem.PorterStemmer()
            tokens =  nltk.tokenize.word_tokenize(text)
            tokens = [stemmer.stem(token) for token in tokens]
            text = " ".join(tokens)
        if punkt_token == 'True':
            tokenizer = PunktSentenceTokenizer()
            tok_sentences = tokenizer.tokenize(text)
            tokens = []
            for s in tok_sentences:
                tokens = nltk.word_tokenize(s, preserve_line =
True)
            text = " ".join(tokens)
        result = text
        texts_results.append(result)
    sentence = pd.Series(texts_results)
    return sentence,labels
```

Python
```
#Classifiers.py

def apply_linear_svc(X_train, y_train, X_test):
    print("----------------")
    print("Linear support vector classifier")
    print("----------------")
    train_array = toNumpyArray(X_train)
    test_array = toNumpyArray(X_test)
    clf = LinearSVC()
```

```python
    clf.fit(train_array, y_train)
    y_predict = clf.predict(test_array)
    return y_predict

def apply_multiple_layer_perceptron(X_train, y_train, X_test):
    print("----------------")
    print("Multiple layer perceptron classifier")
    print("----------------")
    train_array = toNumpyArray(X_train)
    test_array = toNumpyArray(X_test)
    clf = MLPClassifier()
    clf.fit(train_array, y_train)
    y_predict = clf.predict(test_array)
    return y_predict

def apply_k_neighbours(X_train, y_train, X_test):
    print("----------------")
    print("K-neighbours classifier")
    print("----------------")
    train_array = toNumpyArray(X_train)
    test_array = toNumpyArray(X_test)
    clf = KNeighborsClassifier()
    clf.fit(train_array, y_train)
    y_predict = clf.predict(test_array)
    return y_predict

def apply_decision_tree(X_train, y_train, X_test):
    print("----------------")
    print("Decision tree classifier")
    print("----------------")
    train_array = toNumpyArray(X_train)
    test_array = toNumpyArray(X_test)
    clf = DecisionTreeClassifier()
    clf.fit(train_array, y_train)
    y_predict = clf.predict(test_array)
    return y_predict


def apply_random_forest(X_train, y_train, X_test):
    print("----------------")
    print("Random forest classifier")
    print("----------------")
    train_array = toNumpyArray(X_train)
    test_array = toNumpyArray(X_test)
    clf = RandomForestClassifier()
    clf.fit(train_array, y_train)
    y_predict = clf.predict(test_array)
    return y_predict
```

Results

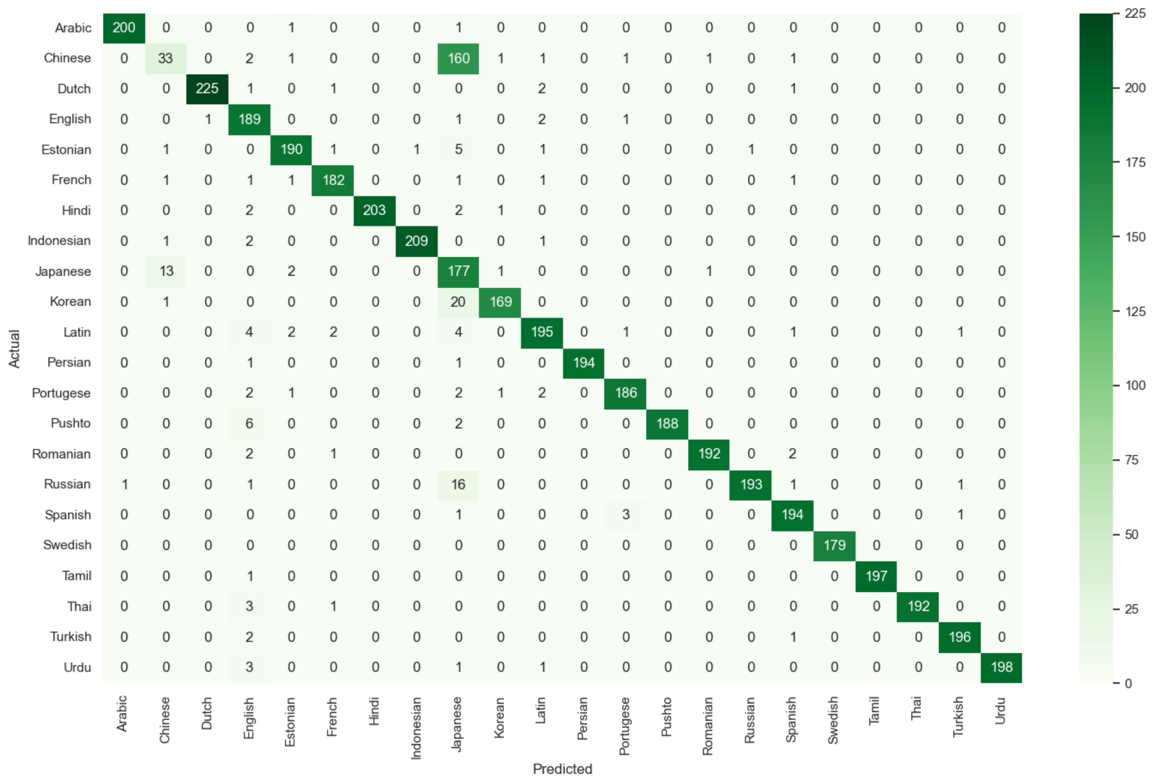
The following table depicts the results of applying the different classifiers.

	NB char	NB words	LSV Char	LSV Words	KNN Char	KNN Words	DT Char	DT Words	RF Char	RF Words	MLP Char	MLP Words
Unig	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000	1000
Mic F1	0.9552	0.8921	0.9659	0.93	0.9543	0.9069	0.9346	0.9043	0.9730	0.9277	0.9718	0.9257
Mac F1	0.9578	0.8812	0.9663	0.9257	0.9550	0.9064	0.9349	0.9012	0.9737	0.9239	0.9723	0.9214
W F1	0.9574	0.8846	0.9662	0.9262	0.9550	0.9069	0.9343	0.9015	0.9734	0.9243	0.9722	0.9219

When we analyze the results for the vocabulary of 1000 characters, we can see that LSV, RF and MLP outperform the Naïve Bayes classifier.
Analyzing the results for the vocabulary of 1000 words, we see that every other classifier used outperforms the Naïve Bayes one.
The best classifier turned out to be the Random Forest irrespective of whether characters or words are used.

Additionally, looking at the confusion matrix (figure 5) when using different classifiers than the base one (Naïve Bayes) and a vocabulary of 1000 words, we noticed that now the misclassification mentioned before (Chinese, Japanese, Korean being classified as Swedish) is no longer present. Now, it mostly misclassifies Japanese as Chinese.

As a side note, we notice that the neural network (MLP) was the one that took the longest to finish.



Actual \ Predicted	Arabic	Chinese	Dutch	English	Estonian	French	Hindi	Indonesian	Japanese	Korean	Latin	Persian	Portugese	Pushto	Romanian	Russian	Spanish	Swedish	Tamil	Thai	Turkish	Urdu
Arabic	200	0	0	0	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
Chinese	0	33	0	2	1	0	0	0	160	1	1	0	1	0	1	0	1	0	0	0	0	0
Dutch	0	0	225	1	0	1	0	0	0	0	2	0	0	0	0	0	1	0	0	0	0	0
English	0	0	1	189	0	0	0	0	1	0	2	0	1	0	0	0	0	0	0	0	0	0
Estonian	0	1	0	0	190	1	0	1	5	0	1	0	0	0	0	1	0	0	0	0	0	0
French	0	1	0	1	1	182	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0
Hindi	0	0	0	2	0	0	203	0	2	1	0	0	0	0	0	0	0	0	0	0	0	0
Indonesian	0	1	0	2	0	0	0	209	0	0	1	0	0	0	0	0	0	0	0	0	0	0
Japanese	0	13	0	0	2	0	0	0	177	1	0	0	0	0	1	0	0	0	0	0	0	0
Korean	0	1	0	0	0	0	0	0	20	169	0	0	0	0	0	0	0	0	0	0	0	0
Latin	0	0	0	4	2	2	0	0	4	0	195	0	1	0	0	0	1	0	0	0	1	0
Persian	0	0	0	1	0	0	0	0	1	0	0	194	0	0	0	0	0	0	0	0	0	0
Portugese	0	0	0	2	1	0	0	0	2	1	2	0	186	0	0	0	0	0	0	0	0	0
Pushto	0	0	0	6	0	0	0	0	2	0	0	0	0	188	0	0	0	0	0	0	0	0
Romanian	0	0	0	2	0	1	0	0	0	0	0	0	0	0	192	0	2	0	0	0	0	0
Russian	1	0	0	1	0	0	0	0	16	0	0	0	0	0	0	193	1	0	0	0	1	0
Spanish	0	0	0	0	0	0	0	0	1	0	0	0	3	0	0	0	194	0	0	0	1	0
Swedish	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	179	0	0	0	0
Tamil	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	197	0	0	0
Thai	0	0	0	3	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	192	0	0
Turkish	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	196	0
Urdu	0	0	0	3	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	198

Figure 5: Confusion matrix using random forest classifier and a vocabulary of 1000 words.

Conclusions

Throughout the laboratory we were able to explore the different preprocessing techniques and classifying algorithms that can be applied when working with the task of language detection.
For the preprocessing part of the laboratory, we tried different techniques that did not improve the results. After analyzing the baseline, we can conclude that the detection by character gives better results than the detection by words. This may be due to the fact that there is a high variety of words compared to the low number of characters.
When working with classifiers it is important to be aware of what we are trying to achieve and the type of data we are working with in order to decide which classifier to use.