



UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Language Detection



Master in Data Science, FIB
Mining Unstructured Data

March 12th, 2023

Pau Comas Herrera - Odysseas Kyparissis

Table of Contents

Introduction	1
Preprocessing Techniques	1
Code	2
Experiments and Results	5
First Baseline	5
Second Part: Preprocessing and Document Structure	7
Conclusion	11

Introduction

The objective of the project is to become familiar with natural language processing (NLP) data and the additional challenges it presents compared to tasks based on structured data. For that, we used a dataset consisting of 21,859 sentences written in 22 different languages, including several language families and writing scripts.

The workload was divided into two main exercises. The first exercise aimed to make us become familiar with unstructured data and natural language, analyzing an already working solution with no preprocessing. Specifically, two systems were compared based on character level granularity and word level granularity, respectively, each with a vocabulary size of 1000 tokens of the most frequent characters or words. We could then analyze the differences in classification results and the reasons behind them.

In the second exercise, the system based on word level granularity was modified and different preprocessing steps and learning models were explored to improve the system's accuracy. Experimentation was done with different vocabulary sizes, as well as sentence splitting, tokenization, punctuation removal, and lemmatization, to understand how different preprocessing steps affected the system's errors and performance. New classifier models (k-Nearest Neighbour, SVM, Decision Tree) were added using frequency features as input to compare their performance with the original naive Bayes classifier.

The report presents the preprocessing steps tried, the rationale and consequences for employing them, and the classifiers used in the experiments. Finally, the report concludes with final remarks and insights gained from the task.

Preprocessing Techniques

Several different preprocessing techniques are applied to the sentences of the dataset before training the classifier models. The approaches tested include preprocessing libraries, like NLTK and SpaCy, that are usually used during natural language processing, in order to achieve different goals on a specific topic. Additionally, due to the fact that the preprocessing steps should have been robust for all 22 different languages, besides these libraries, some heuristic techniques were tested as well.

To begin with, individually using functions of NLTK or SpaCy, sentence segmentation, tokenization, lowercasing, removing of punctuation, and lemmatization techniques were applied to the dataset. For space-saving reasons, only the preprocessing function that makes use of the SpaCy library is included in the report, once it was more robust to the set of languages that were included in the dataset. On the other hand, due to the language-specific application of those libraries, two additional heuristic approaches were tested: *sentence segmentation by the number of words* and *sentence segmentation by character position*. A brief explanation of the

steps is included in this chapter while the experiments and the results are included in the appropriate paragraph.

In greater detail, the main reason for applying sentence segmentation was the idea of splitting complex sentences into individual ones. In that way, we could generate more observations that could be used for training and testing the classifier models. Also, sentence segmentation ensures that each sentence is analyzed separately, which might help to identify language-specific features more accurately. Furthermore, once sentence segmentation was applied, the next step included tokenization and lowercasing of sentences. With tokenization, sentences were split into separate words, on which lowercasing, punctuation removal and lemmatization could be possible to be applied.

Lowercasing is a common text preprocessing step that can help to normalize text data, which makes it easier to compare and analyze. However, in the context of language identification, lowercasing has potential drawbacks such as removing language-specific features, like capitalization of nouns in German. Moreover, the next preprocessing step performed is punctuation removal. In the context of language identification, removing punctuation can introduce probable downsides. Some languages rely more heavily on punctuation than others, and removing punctuation can obscure language-specific features. For example, Spanish uses a distinctive punctuation mark (the inverted question mark) that can be indicative of the language.

Finally, lemmatization is a common preprocessing step that can help to reduce the complexity of text data, by transforming different forms of specific words into the same *lemma*. In that way, the dimensional space that the analysis is performed on can be reduced significantly. However, in the context of language identification, lemmatization has potential drawbacks. Some languages, such as Arabic, rely heavily on inflectional morphology, which means that different forms of the same word can be indicative of the language. In such cases, lemmatization can lead to inaccurate language identification results.

Code

Preprocessing functions:

```
Python
# Load the xx_sent_ud_sm model for sentence segmentation and xx_ent_wiki_sm for lemmatization
nlp_sent = spacy.load('xx_sent_ud_sm')
nlp_lem = spacy.load('xx_ent_wiki_sm')
def preprocess_spacy(sentences, labels):
    # Split the input sentences into individual sentences using the xx_sent_ud_sm model
    preprocessed_sentences = []
    preprocessed_labels = []
    for i,j in zip(sentences.index, labels.index):
```

```

doc = nlp_sent(sentences[i])
for i, sent in enumerate(doc.sents):
    # Lowercase the text
    sent_text_lower = sent.text.lower()
    # Remove punctuation
    sent_text = sent_text_lower.translate(str.maketrans('', '', string.punctuation))
    # Lemmatize the text using the xx_ent_wiki_sm model
    sent_doc = nlp_lem(sent_text)
    sent_lemmas = [token.lemma_ if token.lemma_ != '' else token.text for token in
sent_doc]

    # Strip Lemmas and append the preprocessed sentence and corresponding label to the
results

    preprocessed_sentences.append(' '.join(sent_lemmas).strip())
    preprocessed_labels.append(labels[j])
preprocessed_sentences = pd.Series(preprocessed_sentences)
preprocessed_labels = pd.Series(preprocessed_labels)
# Return the preprocessed sentences and replicated labels
return preprocessed_sentences, preprocessed_labels

```

Python

```

def preprocess_splitting_four_char_based(sentence, labels):
    new_sentences = []
    new_labels = []
    labels_list = labels.tolist()
    num_substrings = 4
    for index, s in enumerate(sentence):
        n, r = divmod(len(s), num_substrings)
        parts = [s[(i * n) + min(i, r):((i + 1) * n) + min(i + 1, r)] for i in
range(num_substrings)]
        for part in parts:
            new_sentences.append(part)
            new_labels.append(labels_list[index])
    new_sentences_series = pd.Series(new_sentences)
    new_labels_series = pd.Series(new_labels)
    return new_sentences_series, new_labels_series

def preprocess_splitting_four(sentence, labels):
    # Split each sentence into exactly 4 parts
    new_sentences = []
    new_labels = []
    labels_list = labels.tolist()
    for index, s in enumerate(sentence):
        words = s.split()

```

```
n = len(words)
parts = []
for i in range(0, n, math.ceil(n / 4)):
    part = words[i:i + math.ceil(n / 4)]
    parts.append(part)
for part in parts:
    new_sentences.append(' '.join(part))
    new_labels.append(labels_list[index])
new_sentences_series = pd.Series(new_sentences)
new_labels_series = pd.Series(new_labels)
return new_sentences_series, new_labels_series
```

Classifiers:

Python

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from utils import toNumpyArray

def applyNearestNeighbour(X_train, y_train, X_test):
    trainArray = toNumpyArray(X_train)
    testArray = toNumpyArray(X_test)
    clf = KNeighborsClassifier()
    clf.fit(trainArray, y_train)
    y_predict = clf.predict(testArray)
    return y_predict

def applySVM(X_train, y_train, X_test):
    trainArray = toNumpyArray(X_train)
    testArray = toNumpyArray(X_test)
    clf = SVC()
    clf.fit(trainArray, y_train)
    y_predict = clf.predict(testArray)
    return y_predict

def applyDecisionTree(X_train, y_train, X_test):
    clf = DecisionTreeClassifier()
    clf.fit(X_train, y_train)
    y_predict = clf.predict(X_test)
    return y_predict
```

Experiments and Results

First Baseline

Table 1. Character level (vocabulary of size 1000)

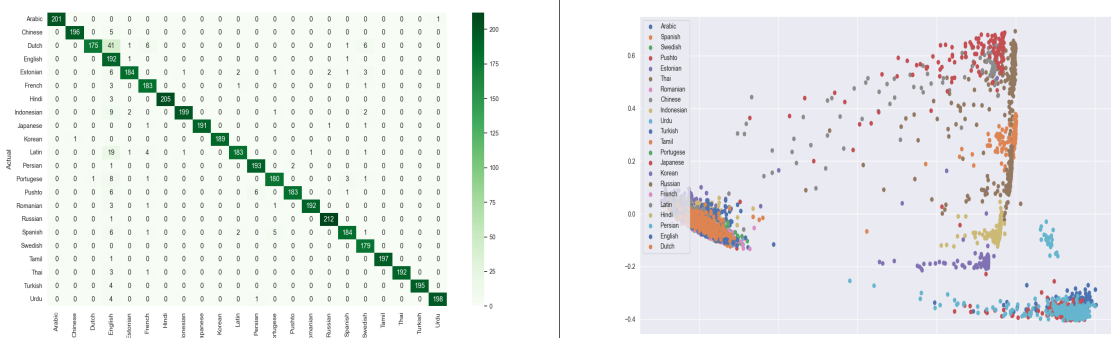
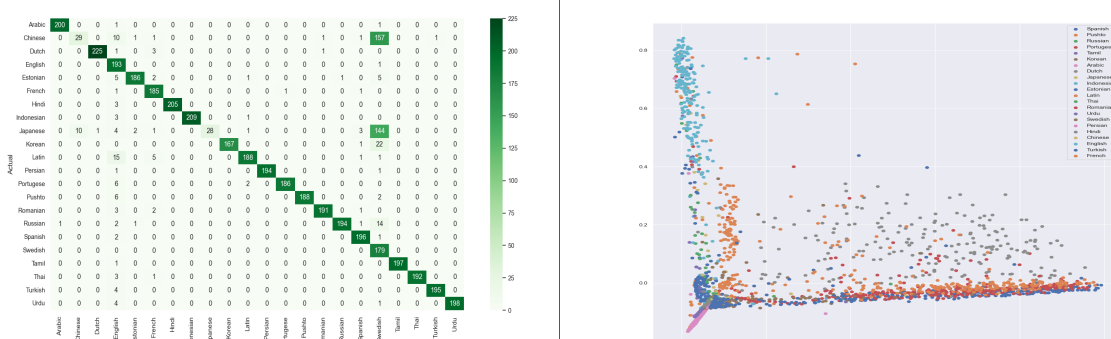
Coverage:	98%
F1-Score (micro)	0.9552
PCA explained variance	[0.3131436 0.13806745]
Confusion matrix and PCA:	
	

Table 2. Word level (vocabulary of size 1000)

Coverage:	26%
F1-Score (micro)	0.89204
PCA explained variance	[0.07878438 0.03638187]
Confusion matrix and PCA:	
	

Comparison:

Coverage at a character level is much bigger than at a word level, being the former roughly at the level of 98% and the latter at 26%. This can be characterized as normal behavior, given the fact that there are much fewer different characters than words in any language. The same is true when comparing between languages. This means that, by taking the most frequent 1000 elements as the application's vocabulary, greater coverage is achieved with characters rather than words.

At a character level, errors in language labeling are sparse, but when they occur, they occur between languages that share the same alphabet. For instance, Spanish and Portuguese are romance languages and that's why we can see for instance 6 Spanish sentences labeled as English or 5 as Portuguese. We can note an important source of errors between English and Latin, with 19 Latin sentences labeled as English. 23 from 26's English letters are in Latin and about 60% of English words are of Latin origin due to borrowing. The most relevant error mistake we had is between Dutch and English, with 41 sentences in Dutch labeled as English. The reason seems to be that both belong to the Germanic language family. This means that they share a common ancestor language and have evolved over time from that shared language.

At a word level, we have much more errors and thus a worse classification (seen in the higher number of confusion matrix values out of the diagonal and the relevant F1 scores). In this case, we have a lot of errors with Chinese, Japanese, and Korean languages being labeled as Swedish. The fact that these three languages are so erratically labeled might not be a coincidence when an approach using a vocabulary based on word tokens is followed.

Chinese, Japanese, and Korean languages often have ambiguous sentence structures, making it challenging to accurately parse and understand the meaning of sentences. For example, Japanese and Korean do not use spaces between words, and the same sequence of characters can have multiple interpretations depending on the context. In addition, Chinese, Japanese, and Korean have limited inflectional morphology, which means that they do not use endings or other grammatical markers to indicate tense, number, or case. Another language with no spaces between words is Thai, but it has 10 times fewer possible headwords than the other three as we consulted on the Internet. Checking our vocabulary space in the application, we could see that over the 1000 most frequent elements, less than 25 in total were Asian, which confirms that the *CountVectorizer* function isn't able to tokenize these languages as well as the others, when creating the matrix of token counts. That means that in training with the token currencies within the 1000 selected words, we have dysfunctional examples about them. Because the tokenizer isn't able to treat these languages properly, the vocabulary of 1000 words doesn't have a meaningful representation of them and the labeling becomes very inaccurate. That's why the application ends up with a lot of them being tagged as Swedish.

The PCA applied to the characters' vocabulary is relatively representative of the data variance, with PC1 representing 31% and PC2 a 13%. We consider that PC1 differentiates the languages

based on their geographical location (related to the alphabet). On the left, we see mostly European and Roman alphabet languages. At around 0.4 we find Asian languages (Korean, Chinese, Japanese, Thai, Tamil...), and on the far right we find Arabic languages such as Arabic, Persian, Turkish and Pashto. The second component (PC2) discriminates between the Asian languages.

The PCA applied to the words' vocabulary is less representative in terms of variance of data, with PC1 and PC2 only adding up to 11%. PC1 value is associated with the sentence being labeled as a romance language or not. We believe that because high values for PC1 are usually for Portuguese, Romanian, Spanish, etc. Differently, in PC2, high values are associated with Germanic languages (English and Dutch). All of the other languages' families or geographic clusters cannot be differentiated in this PCA plot.

Second Part: Preprocessing and Document Structure

Vocabulary size

The word-based vocabulary size determines the size of frequent token features that our model is going to use, given that the training attributes are how many instances of each token a sentence has. This means that we should aim to have the lowest vocabulary size possible to optimize training time but that at the same time we retain a big coverage of the dataset in terms of words.

Char-based coverage was already very high (98%) with a size of 1000, but for words, the coverage decreased substantially to 25-26%. As we can see in Figure 1, which represents the test results we had, coverage follows a first-order transfer function shape, where the growth ratio decreases as we make the vocabulary bigger and bigger, arriving only to 50% with 20 times more words (20000). This is normal given that every time we add a new slot for a word in the vocabulary, it is occupied by a less frequent word. Thus, adding it to the feature space is less meaningful, when vocabulary size is proportionally increasing. That's why in a real application we believe we should choose a manageable size of vocabulary, computational-wise.

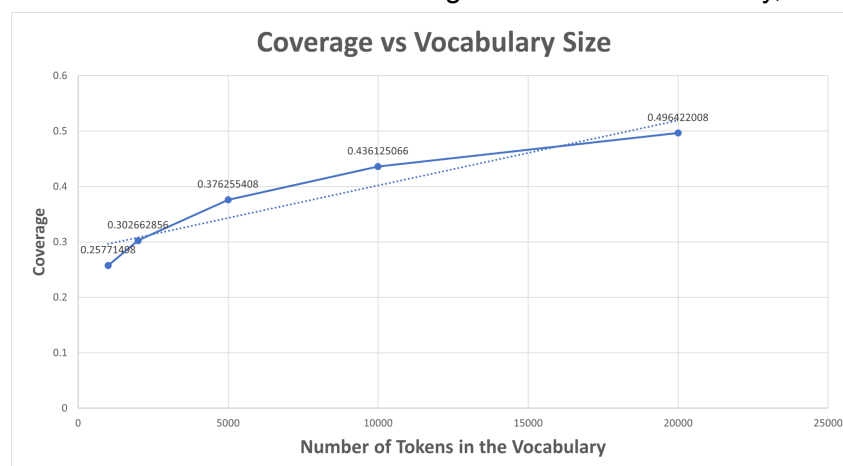


Figure 1. Coverage vs Vocabulary Size

In terms of F1-score (Figure 2) as a metric for the predictive ability of the models (that we'll explain later), testing the vocabulary size for every model we employed, we could see that for Naive Bayes, SVM, and Decision Tree, with default parameters, the improvement is quite small with a bigger vocabulary. At vocabulary size 5000, the increase of F1-score is minimal and thus, having a bigger vocabulary doesn't improve the score anymore. In the case of kNN, we see that maintaining the parameter $k=5$ is only successful for the specific vocabulary size of 1000.

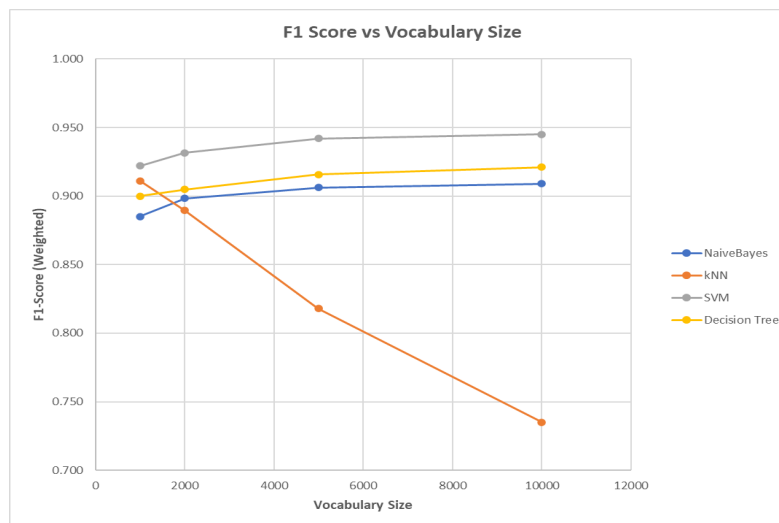


Figure 2. F1-Score vs Vocabulary Size

Sentence splitting

This is a preprocessing technique that we believed we could use regardless of the language, although quality results are not guaranteed. NLTK and SpaCy libraries have already implemented functions that achieve this result. However, those functions are using language-specific techniques to perform correctly. While the language label of each sentence is not available in a language identification application, the following heuristic approaches were tested. Firstly, sentence splitting is done based on words. The text is tokenized by spaces and then we regroup in smaller sentences. This approach assumes that each word is separated by a space, which is true for many languages such as English, Spanish, French, and German. The issue is that, for instance, in our Chinese sentences we didn't even have spaces.

On the other hand, an alternative approach is when sentence splitting is done based on characters. The text is divided into smaller pieces at every occurrence of specific characters, such as a period, exclamation mark, or question mark. The dataset is missing punctuation marks. Consequently, the heuristic used was to divide the sentences by character position, even if it cuts words in half. Besides splitting by character position we still select words as tokens for the training of classifiers. Although less logical at first sight, this approach can be applied to all languages, including those that don't have spaces between words, such as Chinese, Japanese, and Korean, so we wanted to try to see what effect it could have.

We tested splitting our original 17600 sentences into 4 pieces with both sentence splitting techniques and compared results. With sentence splitting based on words, we got 66817 sentences and labels, given that there were languages that couldn't even be split by spaces, as we said. With sentence splitting by char position, we did divide into 70600 sentences and labels.

Table 3. Model with sentence splitting in 4 (by words) (Naive Bayes)

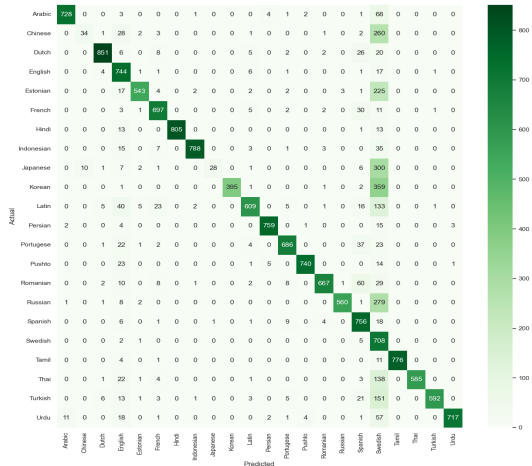
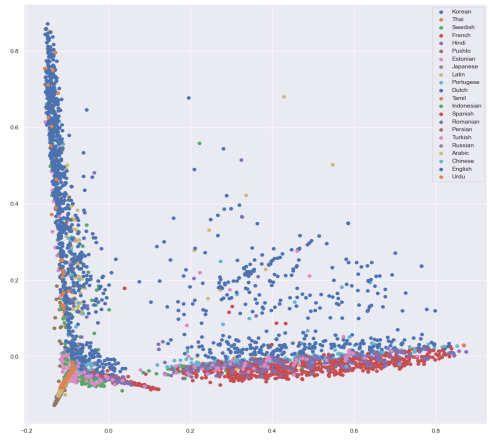
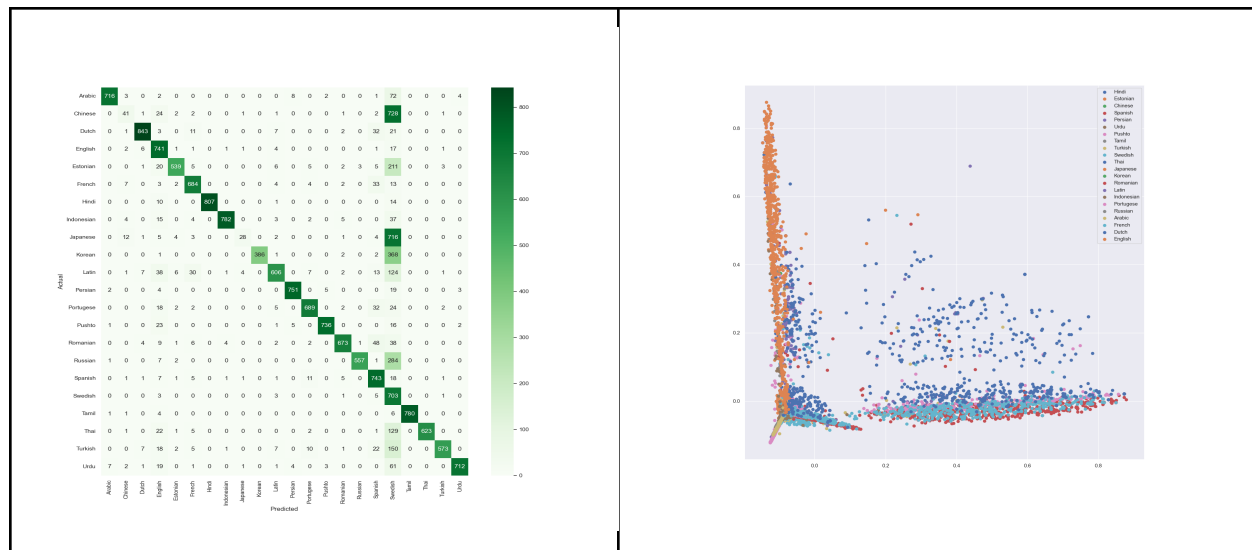
Coverage:	26%
F1-Score (micro)	0.8255
PCA explained variance	[0.05939453 0.02364858]
Confusion matrix and PCA:	
<div style="display: flex; justify-content: space-around; align-items: flex-start;"> <div style="text-align: center;">  <p>Confusion matrix showing Actual vs Predicted for 26 languages. The diagonal elements are highlighted in green, indicating high accuracy. The color scale on the right ranges from 0 to 800.</p> </div> <div style="text-align: center;">  <p>PCA scatter plot showing the separation of 26 languages into distinct clusters. The x-axis ranges from -0.2 to 0.8, and the y-axis ranges from 0.0 to 0.8. A legend on the right identifies the languages by color.</p> </div> </div>	

Table 4. Model with sentence splitting in 4 (by char position) (Naive Bayes)

Coverage:	25%
F1-Score (micro)	0.779
PCA explained variance	[0.05860409 0.02331262]
Confusion matrix and PCA:	



The results above show that generating more data with either of the approaches by splitting sentences didn't translate into help for the models to learn better, and in both cases, our classification quality got worsened. Not only this didn't address the problem that Japanese, Chinese, and Korean had, but added Estonian, Russian, or Thai to the same issue, with a considerable amount of sentences being misclassified as Swedish. Although not included here for extension reasons, a bigger vocabulary (2000 and 5000 tested here) doesn't especially improve any of our preprocessed training results more than what we already analyzed with 17600 sentences unsplit.

Lemmatization

This preprocessing technique, as already introduced, we thought could be useful to synthesize our word bag in a richer way for the creation of the vocabulary features. The problem we encountered is that we realized that we could just use it in a language-specific application. We could use WordNet or alternatives and just lemmatize the words that were included in those dictionaries, but in that case, we are already implying that at least English is present. This is without even mentioning that we have to bear in mind that there are languages with little popularity that cannot be lemmatized in any way, or that don't work with stemmable words. Trying to apply this technique was a good exercise to realize the limitations we had to work on it, so we decided to include the code in this report, where we also tested sentence punctuation and lowercasing, which led us to see that in our dataset they didn't have any effect whatsoever.

Modeling

In order to compare the performance of different models that can be used for language identification applications the following models have been tested: Naive Bayes, k-Nearest Neighbors (kNN), Support Vector Machine (SVM), and Decision Tree. As the main idea of this lab assignment was not to deeply dive into model characteristics and tuning, their basic versions were used. By that, we mean that hyperparameter tuning was not conducted, but only the

default values were used for the parameters of each individual model. Briefly, Naive Bayes is a probabilistic machine learning model that is used for classification tasks. It is based on Bayes' theorem and assumes that all features are independent of each other. Naive Bayes works by calculating the probability of a given data point belonging to each class based on the features it possesses. It then chooses the class with the highest probability as the predicted class. In kNN, the algorithm determines the k nearest data points to a given test data point and predicts its class based on the majority class among its k nearest neighbors. The value of k is determined by the user and is typically chosen based on cross-validation performance. For a vocabulary size of 1000, k=5 was our optimal value. Additionally, SVM is a powerful machine learning algorithm that works by finding the optimal hyperplane that maximizes the margin between the classes in the data. The hyperplane that maximizes the margin is chosen as the decision boundary between the classes. SVM produced the best results in all cases. Finally, a decision tree is a machine-learning algorithm that recursively partitions the feature space into smaller regions based on the values of the features. The algorithm chooses the best feature to split the data based on an impurity measure, such as entropy or Gini impurity. The algorithm continues this process until all the data points belong to the same class or until a stopping criterion is met.

Conclusion

This project allowed us to explore various preprocessing techniques for Natural Language Processing and models, enabling us to analyze the impacts of each approach. Through our experimentation, we discovered that differences between languages can make it very challenging to create a generic application for language detection at the word level, compared to the character level. Therefore, we learned it is important to consider the unique characteristics of each language when applying NLP techniques at determining feature data for the models we choose to apply and test.

Furthermore, this project also provided us with a better understanding of the differences between languages geographically and the similarities and differences between language families. By examining the performance of various NLP techniques and models on languages from different language families, we were able to gain insight into the unique characteristics and challenges posed by them.