UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

# MINING UNSTRUCTURED DATA
# Document Structure and Language Detection

MUD Course Project

March 14, 2022

**Daniel Arias**
**Mohana Fathollahi**

# 1 Introduction

In this project we want to detect 22 different languages. We will apply different approaches to deal with text and compare performance of each of them. Next, we will analyze various preprocessing method and compere their result. Moreover, different algorithms for classification will be applied.

# 2 First Baseline

In this part we want to compare different approaches to detect various languages and answer some questions related to each scenario. For this part 1000 tokens have been used and to compare models we used F1-micro for all part of this project.

## 2.1 Tokenization

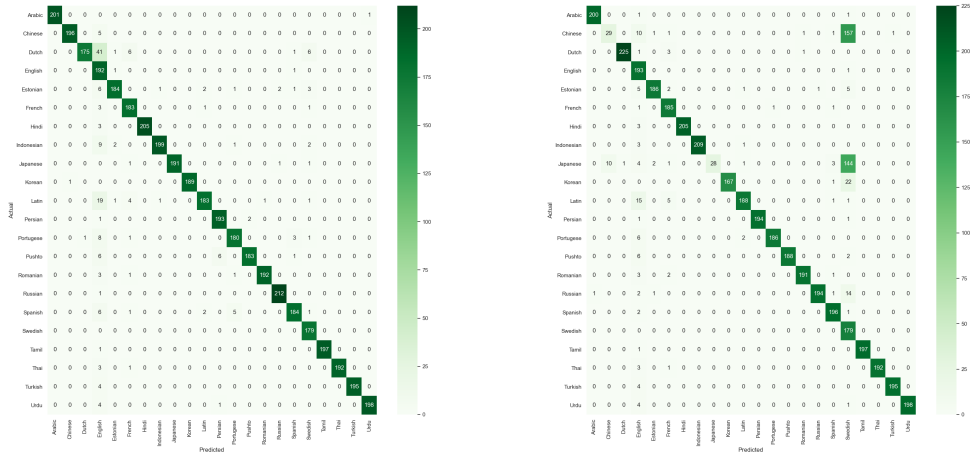
How well does the vocabulary covers the data?
We used word and character as tokens, in the table below we compare result of two approaches. Based on this result, character covers 98 percentage but word cover just 26 percentage.
To get higher coverage we should use more than 1000 words. In the section 3 and vocabulary size we will see effect of different number of tokens in coverage. Additionally, when we use character as token we can get high accuracy and we can explain higher variance with first two PCs.

| Token | Coverage | F1 | PCA |
|---|---|---|---|
| Word | 0.258 | 0.892 | [0.07878438, 0.03638178] |
| Char | 0.981 | 0.96 | [0.3131436, 0.13806745] |

Table 1: Comparing word and character

Which languages produce more errors? What do they have in common (family, script, etc)?
Based on figure 1b, when we use word as a token, number of miss classification in two language are considerable. In chines 78% items classified as a Swedish and just 14% items correctly classified.
Additionally, in Japanese we have same behaviour and approximately 14% truly classified and 74% considered as Swedish.
We can conclude that, because languages such as Chinese, Japanese and Korean are based on symbols, using character is more helpful than word to detect these languages.
On the other side, when we use character we can find some miss classification in languages with approximately same alphabetic such as European languages, for example in Dutch 18% classified as Dutch and 76% considered as English.
for this kind of languages using word gave us better result, because some characters are common between these languages but combination of characters, word, will be different in each one of them [3].

How languages overlap on the PCA plot? What could that overlapping mean?
There are overlaps in some languages, specially languages that have same alphabetic. For example Persion, Arabic, Urdu and Pushto based on plot 2 have overlaps, consider as first group. In the left side of plot, we have European languages, such as; Portuguese, Spanish, Latin, English, Dutch and Swedish, consider as second group. In the second group, because they have the same root, there are overlaps between them. Moreover, first group has more correlation with first PC and second group has more correlation with second PC.



(a) Character (b) Word

Figure 1: Confusion Matrix


Figure 2: PCA for character

## 2.2 n-Gram

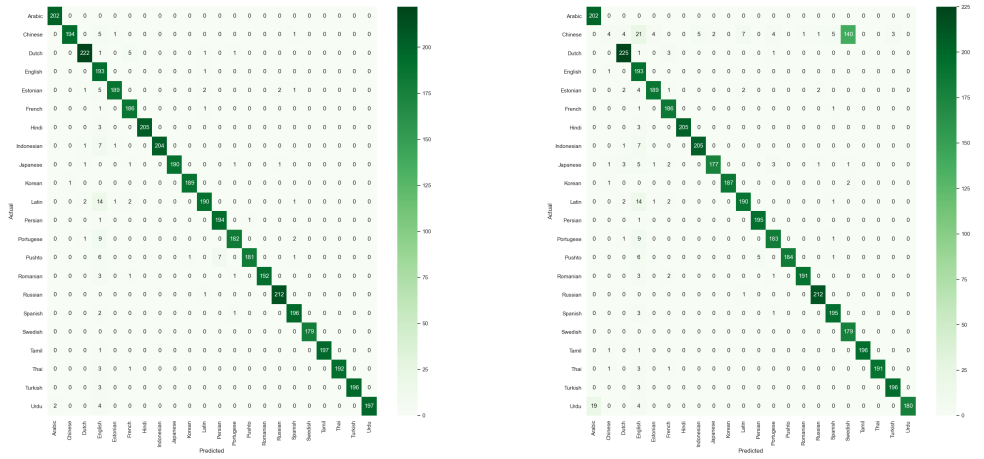
For this part we will compare unigram, bigram and mix of unigram and bigram in both approaches, word and charceter. Based on table below, when we use character as token, mix of unigram and bigram will give us higher accuracy compare to other two approaches.

As we can see in the figure 3b, when we used bigram, in Chinese language just 2% is correctly

| | Formal Definition | coverage | F1 | PCA |
|---|---|---|---|---|
| unigram | $ngramrange(1,1)$ | 0.981 | 0.955 | [0.3131436, 0.13806745] |
| bigram | $ngramrange(2,2)$ | 0.0 | 0.924 | [0.17544056, 0.06881078] |
| unigram and bigram | $ngramrange(1,2)$ | 0.934 | 0.973 | [0.2882978, 0.13025099] |

Table 2: Comparing different grams for character

classified. While, when Mix of unigram and bigram has been used number of miss classification significantly decreased and we could get better accuracy compare to using just unigram.



(a) Mix of unigram and bigram

(b) Bigram

Figure 3: Confusion Matrix-bigram and mix of unigram and bigram for character

Same steps have been done for words, in the table below we can see the final result of each gram. The lowest accuracy belong to bigram, therefore to deeply analyze the behaviour of it, we should use confusion matrix.

| | coverage | F1 | PCA |
|---|---|---|---|
| unigram | 0.981 | 0.825 | [0.05864405, 0.02522933] |
| bigram | 0.0 | 0.48 | [0.03206978, 0.0221104] |
| unigram and bigram | 0.41 | 0.834 | [0.05563759, 0.02453338] |

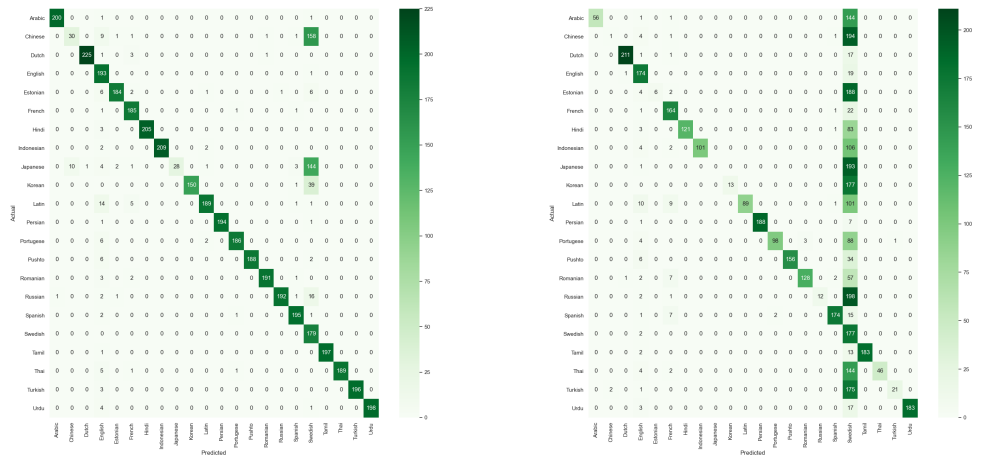
Table 3: Comparing different grams for word

Based on confusion matrix in figure 4b number of miss classification in bigram in most of languages is more than miss classifications in mix of unigram and bigram.

# 3 Preprocessing and Document Structure

## 3.1 Vocabulary size

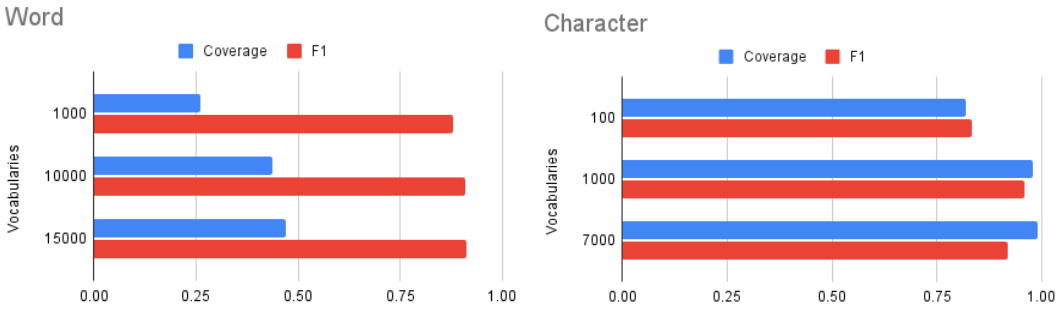
As we saw in the previous part, when we used 1000 words we could not get high coverage. Therefore, different values have been test to find a better coverage. Based on figure 5b when number of words or character have been increased coverage will increase too.



(a) Mix of unigram and bigram

(b) Bigram

Figure 4: Confusion Matrix-bigram and mix of unigram and bigram for word




(a) Word

(b) Character

Figure 5: Compare different amount of vocabularies

## 3.2 Preprocessing

As general preprocessing we used regular expressions for removing numbers and special characters from the inputs. Additionally, for preprocessing we try different steps to train the classifier, such as tokenization [7], sentence splitting, lemmatization and steamming [6].

- Tokenization: It divides strings into lists of sub-strings. We use simple tokenization because we are dealing with different languages and the punctuation could not perform well for those. From these strings divided by the tokenization we remove the ones that are stop words.
- Sentence splitting: This is the process of splitting text into individual sentences adding more data to train the data set.
- Lemmatization: This is the process that group together words with the same root or lemma form for each input word but with different inflections or derivatives of meaning so they can be analyzed as one item. We are aware that Lematization supports just English language, but based on data set that we have, there are some combination of English and other languages, therefore, applying Lemmatization can be helpful in these cases.
- Stemming: This is a natural language processing technique that lowers inflection in words to their root forms, hence aiding in the preprocessing of text, words, and documents for text normalization. For Stemming we used the 'SnowballStemmer' as it can map non-English words.

### 3.3 Lemmatization vs Stemming

The goal of both stemming and lemmatization being special cases of normalization is to scale back inflectional forms associated with a word to a standard base form. Stemming algorithms work by keeping apart the tip or the start of the word, taking under consideration a listing of common prefixes and suffixes that may be found in an inflected word, this approach may lead to errors. On the opposite hand, Lemmatization takes into consideration the morphological analysis of the words, for this, it's necessary to possess detailed dictionaries which the algorithm can check to link the shape back to its lemma.

### 3.4 Languages supported

The different 22 languages used in the laboratory are: ['russian', 'urdu', 'japanese', 'dutch', 'indonesian', 'chinese', 'persian', 'latin', 'turkish', 'portugese', 'spanish', 'english', 'french', 'arabic', 'estonian', 'pushto', 'thai', 'hindi', 'swedish', 'romanian', 'korean', 'tamil']

- Stemming: from the 'nltk.stem' library we downloaded the 'wordnet' lexical database. The SnowballStemmer 16 languages supported are: ['arabic', 'danish', 'dutch', 'english', 'finnish', 'french', 'german', 'hungarian', 'italian', 'norwegian', 'porter', 'portuguese', 'romanian', 'russian', 'spanish', 'swedish']
- Stop Words: from the 'nltk.corpus' the 'stopwords' module support the following 24 languages: ['arabic', 'azerbaijani', 'bengali', 'danish', 'dutch', 'english', 'finnish', 'french', 'german', 'greek', 'hungarian', 'indonesian', 'italian', 'kazakh', 'nepali', 'norwegian', 'portuguese', 'romanian', 'russian', 'slovene', 'spanish', 'swedish', 'tajik', 'turkish']

The Stop words languages that match with the ones in the data-set are 10. In the case of Stemming only 8 out of 22. Meaning that for these preprocessing methods the ratio is not even 50% which could lead to misclassifications.

### 3.5 Code

```
def preprocess(rows, labels, method):
    [...]
    # rows is a series -> {index, value}
    for index, row in rows.items():

        row = GeneralPreprocess(row)
        label = labels[index].lower()
        return_labels[index] = label

        if method == Method.LEMATIZATION._value_:
            Lematization(lemmatizer, row, preprocessSentences, index)
        if method == Method.STEAMING._value_:
            Stemming(supported_stemm_langs, row,
                     label, preprocessSentences, index)
        if method == Method.SENTENCE._value_:
            max_i = SentenceTokenization(
                return_labels, preprocessSentences, max_i, row, label, index)
        if method == Method.TOKENIZATION._value_:
            RemoveStopWords(stop_words_by_lang, row, label,
                            stop_words_keys, preprocessSentences, index)
        if method == Method.NOTHING._value_:
            preprocessSentences[index] = row

    # return series of sentences {index, value}, series of labels {index, value}
    return pd.Series(preprocessSentences), pd.Series(return_labels)

```

```python
def SentenceTokenization(return_labels, preprocessSentences, max_i, row, label,index):
    # Sentence Tokenize
    sents_words = nltk.sent_tokenize(row)
    preprocessSentences[index] = row
    if len(sents_words) > 1:
        for sentence in sents_words:
            preprocessSentences[max_i] = sentence
            return_labels[max_i] = label
            max_i += 1
    return max_i

def RemoveStopWords(stop_words_by_lang, row, label, stop_words_keys, preprocessSentences, index):
    filtered = []

    if label in stop_words_keys:
        words = nltk.word_tokenize(row)
        for w in words:
            if not w in stop_words_by_lang[label]:
                filtered.append(w)
    if filtered:
        preprocessSentences[index] = ' '.join(
            map(str, nltk.word_tokenize(row))).strip()
    else:
        preprocessSentences[index] = row


def GeneralPreprocess(row):
    # To lower, remove special characters, numbers, words with - and '
    return row.lower().replace(r'\d+', '').replace(r'[^A-Za-z0-9]+', '')
                                          .replace(r"(?:[a-z][a-z'\-_]+[a-z])",  '')


def Lematization(lemmatizer, row, preprocessSentences, index):
    words = nltk.word_tokenize(row)
    lemmatize_words = [lemmatizer.lemmatize(word) for word in words]
    preprocessSentences[index] = ' '.join(map(str, lemmatize_words)).strip()


def Stemming(supported_stemm_langs, row, label, preprocessSentences, index):
    if label in supported_stemm_langs:
        stemmed = [SnowballStemmer(label).stem(word)
                   for word in nltk.word_tokenize(row)]
        preprocessSentences[index] = ' '.join(map(str, stemmed)).strip()
    preprocessSentences[index] = row
```

Whole project code could be found in [1].

# 4 Experiments and results

## 4.1 Analyze prepocessing approaches

In here we observed the variance between the results given by running the program with different parameters 'Voc.Size'.

### 4.1.1 Coverage

First, we will compare coverage for different vocabulary size and different preprocessing approaches. Based on table below, when number of words increased from 1000 to 15000 coverage increased more than doubled. Moreover, we can see that Lematization can cover more words compare to other approaches in most cases, except when we used 15000 words that Tokenization can reach to better coverage compare to Lematization.

| Voc. Size | GENERAL | LEMATIZATION | SENTENCE | STEAMING | TOKENIZATION |
|---|---|---|---|---|---|
| 1000 | 0.257715 | 0.263536 | 0.257267 | 0.257715 | 0.260797 |
| 1500 | 0.284003 | 0.290400 | 0.283292 | 0.284003 | 0.287280 |
| 2000 | 0.302663 | 0.309980 | 0.302233 | 0.302663 | 0.306257 |
| 2500 | 0.319916 | 0.327987 | 0.319428 | 0.319916 | 0.323786 |
| 3000 | 0.334560 | 0.343225 | 0.334106 | 0.334560 | 0.338571 |
| **10000** | 0.436125 | **0.448340** | 0.435411 | 0.436125 | 0.441853 |
| 15000 | 0.471449 | 0.477950 | 0.471057 | 0.471449 | 0.484760 |

Table 4: Compare of coverage for different approaches and vocabulary size

## 4.2 Variance Explained

In this part, we will compare variance that can be explained in first two PCAs. Based on the table section 4.1.1 Lematization can explained higher variance in first dimension compare to other methods.

| Voc. Size | Dimension | GENERAL | LEMATIZATION | SENTENCE | STEAMING | TOKENIZATION |
|---|---|---|---|---|---|---|
| 1000 | 1 | 0.07878438 | 0.08223753 | 0.07850653 | 0.07878438 | 0.07872592 |
| | 2 | 0.03638178 | 0.03605507 | 0.03623967 | 0.03638187 | 0.0363928 |
| 1500 | 1 | 0.07470635 | 0.07804825 | 0.07452338 | 0.07470635 | 0.07465298 |
| | 2 | 0.03403156 | 0.03332383 | 0.03395181 | 0.03403157 | 0.03401257 |
| 2000 | 1 | 0.07229505 | 0.07556224 | 0.07209638 | 0.07229505 | 0.07230818 |
| | 2 | 0.0324767 | 0.03170253 | 0.03239066 | 0.03247665 | 0.03247373 |
| 2500 | 1 | 0.07053935 | 0.07361983 | 0.07033769 | 0.07053935 | 0.07048597 |
| | 2 | 0.03133016 | 0.03051307 | 0.03124934 | 0.03133016 | 0.03135374 |
| 3000 | 1 | 0.06884188 | 0.07194157 | 0.06862428 | 0.06884188 | 0.06884146 |
| | 2 | 0.0303541 | 0.02964263 | 0.03033824 | 0.03035414 | 0.03034831 |
| **10000** | 1 | 0.05864405 | **0.06146457** | 0.05843296 | 0.05864405 | 0.05863936 |
| | 2 | 0.02522936 | **0.02452843** | 0.02518855 | 0.02522936 | 0.02522068 |
| 15000 | 1 | 0.05555206 | 0.0584053 | 0.05536588 | 0.05555206 | 0.05555217 |
| | 2 | 0.02379828 | 0.02319559 | 0.02377259 | 0.02379828 | 0.0237806 |

Table 5: Comparing preprocessing methods alongside different Vocabulary Size

## 4.3 F1 Score

At the end, we compared F1 score for different methods in table section 4.3. When we used 1000 words Lematization has higher accuracy, while when we increased vocabulary size accuracy did not change

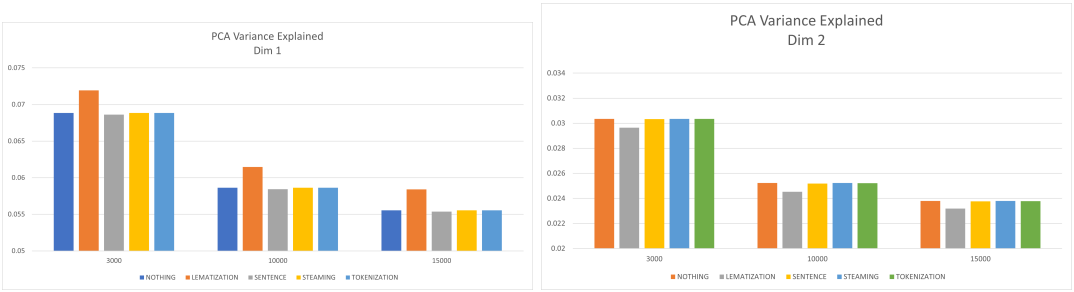
after applying different preproccessing methods.

| Voc. Size | GENERAL | LEMATIZATION | SENTENCE | STEAMING | TOKENIZATION |
|---|---|---|---|---|---|
| 1000 | 0.892045455 | 0.892727273 | 0.891758365 | 0.892045455 | 0.892045455 |
| 1500 | 0.900227273 | 0.899772727 | 0.899842803 | 0.900227273 | 0.900227273 |
| 2000 | 0.903409091 | 0.903409091 | 0.903211318 | 0.903409091 | 0.903409091 |
| 2500 | 0.905 | 0.904545455 | 0.904558724 | 0.905 | 0.905 |
| 3000 | 0.908409091 | 0.908409091 | 0.908151808 | 0.908409091 | 0.908409091 |
| **10000** | 0.915454545 | **0.915454545** | 0.915113407 | 0.915454545 | 0.915454545 |
| 15000 | 0.918636364 | 0.918409091 | 0.918032787 | 0.918636364 | 0.918636364 |

Table 6: Comparing accuracy for different approaches alongside different vocabulary size

## 4.4 Visualisation

In figure fig. 6 we can observe how the amount of Explained Variance augment as the vocabulary size used is higher. We started with 3000 until 15000. We notice that the variance explained is inversely related with the vocabulary size.

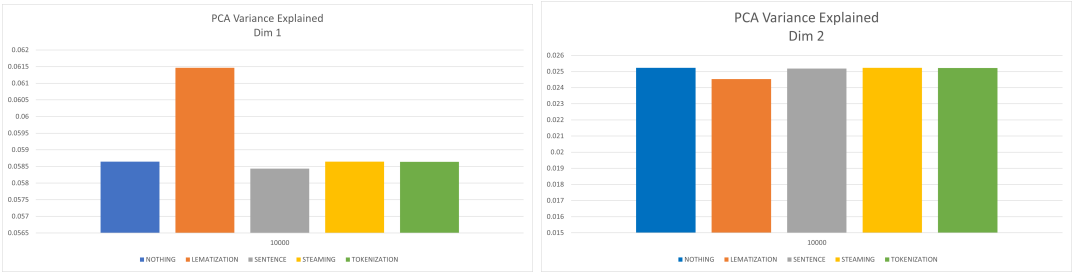



(a) PCA Dim 1 Voc.Size 3000 to 15000

(b) PCA Dim 2 Voc.Size 3000 to 15000

Figure 6: PCA Variance Explained Voc.Size 3000 to 15000

After we check which preprocess method performs better in the 15000 range. Results can be observed in fig. 7. In this case Lemmatization stands out over the others preprocessing methods although the difference with others in average is just **0.009**. However it was not the same for dimension 2 where it is slightly behind the other preprocessing methods.



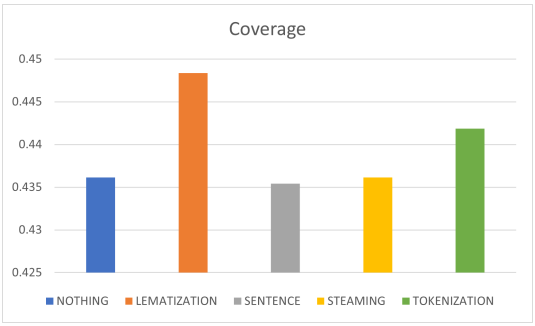

(a) PCA Dim 1 Voc.Size 10000

(b) PCA Dim 2 Voc.Size 10000

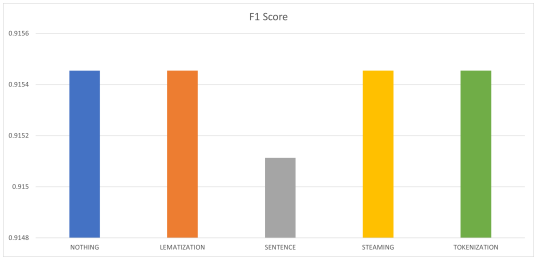
Figure 7: PCA Variance Explained Voc.Size 10000

In the case of the F1 Score Lemmatization has the same result as the others, in contrast with coverage, where again it perfomed marginally better illustrated in fig. 8 .

Therefore we selected Lemmatization as the best preprocess method for in order to make the comparison between the different classifiers with Word or Character granularity level.




(a) Coverage Voc.Size 10000




(b) F1 Score Voc.Size 10000

Figure 8: Coverage and F1 Score Voc.Size 10000

## 4.5 Classifiers

In this project, we used five different kind of classifiers and we will compare result of each of them in below tables.

- Naive Bayes: This algorithm is based on Bayes' Theorem with an assumption of independence among predictors. It calculates the probability of each tag for a given text and then output the tag with the highest one [5].
- k Nearest Neighbour: We will identify the K nearest neighbors which has the highest similarity score among the training corpus. K denotes how many closest neighbors will be used to make the prediction and we used 5 neighbors and similarity between neighbor is based on Manhattan distance. We get best performance with Manhattan distance [2].
- Logistic Regression: For this method, we used Saga solver because it is faster for large data set. Furthermore, to adjust wights of each class balanced has been used for class weight [4].
- Support Vector Machine: It aims to find an optimal boundary between the possible outputs. The objective is to find a hyperplane that maximizes the separation of the data points to their potential classes in an n-dimensional space [9].
- Random Forest: Estimates is based on the combination of different decision trees. Effectively, it fits a number of decision tree classifiers on various sub samples of the dataset [8].

### 4.5.1 Analyze different classifiers

In this part, we will compare coverage, variance explained and F1 score for above classifiers. We used 10000 vocabulary size, we could not run the all models for higher amount of vocabulary size because of limited RAM in our PC.

Based on result of table section 4.5.1, we can get high coverage when we used character, as mentioned before. Additionally, coverage for different classifiers will be the same.

| Granularity | Voc.Size | NB | KN | SVM | LR | RF |
|---|---|---|---|---|---|---|
| word | 10000 | 0.44834025 | 0.44834025 | 0.44834025 | 0.44834025 | 0.44834025 |
| char | 10000 | 0.99966678 | 0.99966678 | 0.99966678 | 0.99966678 | 0.99966678 |

Table 7: Coverage Classifiers Voc.Size 10000

Result of table section 4.5.1, explained amount of variance for each of algorithms for first two dimensions. As same as coverage behaviour, explained variance for all methods did not change and with character we can explained more variance compare to word.

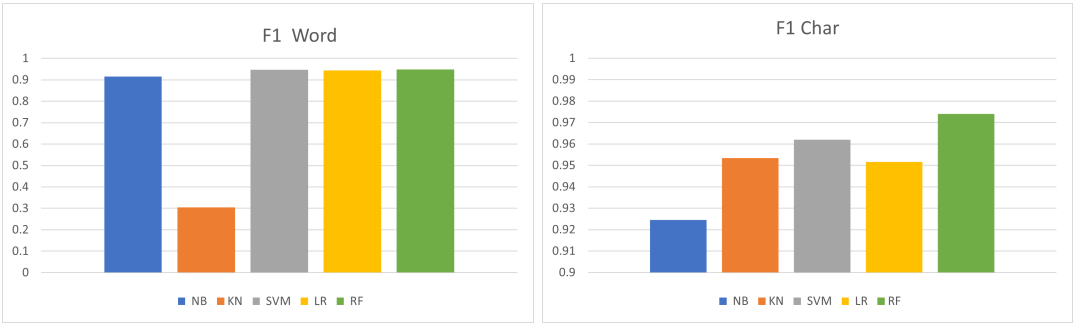
| Granularity | Dimension | Voc.Size | NB | KN | SVM | LR | RF |
|---|---|---|---|---|---|---|---|
| word | 1 | 10000 | 0.06146457 | 0.06146457 | 0.06146457 | 0.06146457 | 0.06146457 |
| char | 1 | 10000 | 0.31197299 | 0.31197299 | 0.31197299 | 0.31197299 | 0.31197299 |
| word | 2 | 10000 | 0.02452842 | 0.02452845 | 0.02452843 | 0.02452845 | 0.02452844 |
| char | 2 | 10000 | 0.13559094 | 0.13559094 | 0.13559094 | 0.13559094 | 0.13559094 |

Table 8: Classifiers PCA Variance Explained Voc.Size 10000

In the table below, we have accuracy for each algorithm with respect to the best preprocessing method 'Lemmatization'. Based on this table, Random Forest gives us higher accuracy in both character and word. As we concluded in the first part, accuracy when we use character is higher than when we use word. But in this part difference between these two approach is less than before, because of two reasons: applying good classifier and using appropriate preprocessing. In the figure section 4.5.1, comparison between models, visually has been provided.

| Granularity | Voc.Size | NB | KN | SVM | LR | RF |
|---|---|---|---|---|---|---|
| word | 10000 | 0.915454545 | 0.304772727 | 0.947272727 | 0.944090909 | 0.948409090 |
| char | 10000 | 0.924545455 | 0.953409091 | 0.962045455 | 0.951590909 | 0.974090909 |

Table 9: F1 Classifiers Voc.Size 10000




(a) Classifiers F1 Score 'Word'

(b) Classifiers F1 Score 'Character'

Figure 9: Compare accuracy for different algorithms

# 5 Conclusions

Based on our analyses, we can find that when we used character as a token, we could get higher accuracy in most of languages. While when we used words, we could not get good accuracy in Chines and Japanese languages and after using different preprocessing methods, the result did not improve significantly. The main reason is that these languages are not supported in the different preprocessing methods provided by nltk as we saw in previous sections of the preprocessing.
On the other side our approach to classify languages were based on traditional machine learning algorithms, if we use deep learning approaches we may get higher accuracy even words have been used as a token.

# References


[1] *GitHub address*. URL: https://github.com/daniel-arias07/Lab1_MUD_MDS_UPC.

[2] *k-nearest-neighbors*. URL: https://towardsdatascience.com/machine-learning-basics-with-the-k-nearest-neighbors-algorithm-6a6e71d01761#:~:text=Summary,that%20data%20in%20use%20grows..

[3] *Kaggle competition in language detection*. URL: https://www.kaggle.com/martinkk5575/language-detection/notebook.

[4] *Logistic regression*. URL: https://en.wikipedia.org/wiki/Logistic_regression.

[5] *Naive Bayes*. URL: https://en.wikipedia.org/wiki/Naive_Bayes_classifier.

[6] *NLTK :: nltk.stem.wordnet*. URL: https://www.nltk.org/_modules/nltk/stem/wordnet.html.

[7] *NLTK :: nltk.tokenize package*. URL: https://www.nltk.org/api/nltk.tokenize.html.

[8] *Random Forest*. URL: https://en.wikipedia.org/wiki/Random_forest.

[9] *Support Vector Machines*. URL: https://scikit-learn.org/stable/modules/svm.html.