Deliverable I: Language Detection

Course Project - Mining Unstructured Data

Enric Reverter
Louis Van Langendonck
Facultat d'Informàtica de Barcelona

2022
Universitat Politècnica de Catalunya

Contents

1 Introduction

The goal of this report is to analyze and adapt a language classifying pipeline in order to gain insight and familiarity to typical Natural Language Processing (NLP) tools and strategies. To do so, the following structure is maintained. First, in Section 2, a baseline analysis is done. It describes the given pipeline by reporting on its performance by changing parameters like the vocabulary size and the unigram granularity. Next, in Section 3, different possible preprocessing steps are proposed. It is described how they function and if they may or may not change results. The results of such steps are interpreted throughout the section. Thirdly, in section 4, different classifiers are put forth, while again pointing out the possible effects these may have. Again, their results are interpreted along their descriptions. Finally, in Section 5, concluding remarks are made. Throughout the document, relevant python code will be included and integrated.

2 First Baseline

2.1 Character level granularity & 1000 character vocabulary size

For this first case, the pipeline yields very accurate predictions. First of all it has a coverage of about 98 %. This is very high, but expected as any piece of text is way more likely to contain about all most occurring characters of that language than all words. The confusion matrix of the test set predictions shows significantly accurate results. For every language, the most frequent prediction is the right one. The most noticeable mistake is made between Dutch and English. This is not too much of a surprise as these languages completely share their alphabet. The next most noticeable mistake is between English and Latin. These again share their alphabet. Now, the PCA plot is looked into. Together, the two dimensions cover about 45 % of the variance in the training set. Because these are only two dimensions, the languages are not at all perfectly separable, however, some distinctions can be made. Generally, three clusters can be loosely identified: a cluster on the left, representing mostly Latin alphabet languages, a cluster at the bottom right, representing middle-eastern languages and lastly the rest contains well separated Asian and other non-Latin languages. Finally, the weighted F-1 score of 95.74 % confirms the accuracy of the predictions.

2.2 Character level granularity & 10000 character vocabulary size

This second case increases the character vocabulary size. It can immediately be noted that only 6816 character are included, which means that all char-

acters in the training set are included. The coverage of 99,97 % is therefore basically complete. However, the confusion matrix shows the algorithm performs worse. It now misclassifies Dutch, English, Swedish and Latin more than before. It is not completely clear why this happens. Most likely, including all characters, the languages have more similar occurring characters. However, knowing for example Dutch and Swedish, they contain quite some different characters such that a part of the problem might also be because of low quality or faulty training data. The PCA dimensions shows very similar results and coverage as in 2.1. The weighted F1-score unsurprisingly yields a good but lower result of about 92 %.

2.3 Word level granularity & 1000 word vocabulary size

For word granularity some significantly different results are found. The coverage is reduced to 25.77 %. This is to be expected because the amount of distinct words in a language is significantly higher than the amount of distinct characters. Moreover, the vocabulary only includes 1000 words, which is little in order to cover all languages in the set. The confusion matrix, however, shows that the pipeline doesn't perform too bad although some misclassifications happen very often: Mostly Chinese and Japanese are not recognized and classified as Swedish instead. This can be depicted in 3.2. That these languages get misclassified is to be expected, as the algorithm tokenizes the sentences in these languages as words because there are no spaces between the different characters, each resembling a word. Consequently, as the chance of having a reoccurring sentence is very small, it often fails to correctly classify these languages. But why it is classified as Swedish instead? This might be because of the Bayes classifier opting for the language with the highest prior probility, which would be the most occurring language. Although all languages are equally represented in the data set, after the train-test split, some imbalance may be present, thus possibly favoring swedish. However, these ideas are considered just as an hypothesis. Moving on, the PCA dimensions now together cover only about 11,5 % of the variance. This low amount is visible on the actual plot as it is quite difficult to distinguish languages and clusters. A very loose analysis might indicate that the first dimension distinguishes between Latin and non-Latin alphabets, with the non-Latin alphabet languages at low values. The second dimension seems to mostly contain European languages at low values. The weighted F-1 score of 88.46 % confirms that the pipeline performs fairly well but worse than with character granularity.

2.4 Word level granularity & 10000 word vocabulary size

Now the word vocabulary size is increased. This results in an increase of coverage to about 43.6 %. This is to be expected, as the ten-fold of words are

now included in the training set. However, also as expected, it is still significantly lower than character granularity. Looking at the confusion matrix, the classification is better than with the smaller vocabulary size, however, it still struggles with the same problems, namely, misclassifying Chinese and Japanese. However, it performs a slight bit better as probably some often reoccurring sentences in these languages might be included in the vocabulary now, however, they most likely still contribute only a minor amount. The PCA plot is very similar to the one in 2.3, although the variance coverage is reduced to only a combined percentage of about 8.5 %. The weighted F-1 score of about 91 % confirms the slight improvement in classification accuracy.

3 Preprocessing

Different preprocessing strategies are now inserted into the pipeline. These have to be applied before the unigram vectorizer, independent from the language label. The different explored possibilities are described below, accompanied by the corresponding code, even if a method ultimately did not deem useful. The performance of the baseline pipeline can be depicted below.

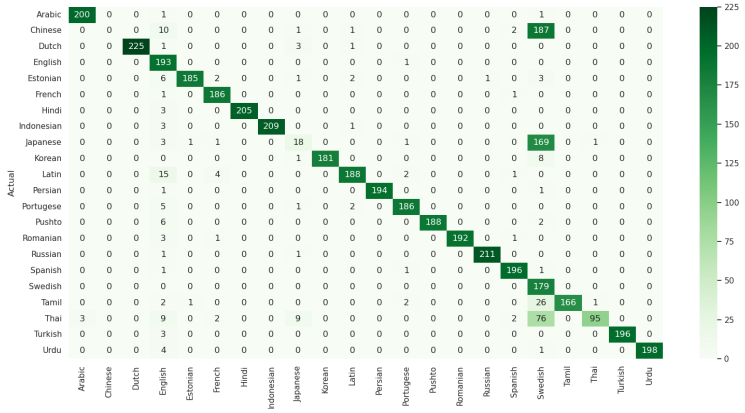

Figure 3.1: Confusion matrix of the baseline classification with 1000 words and the Naive Bayes classifier.

3.1 Improved Tokenization

Tokenization splits text in smaller 'token' units. In order to do more than just split words by spaces, more functional tokenizing methods are used. The *RegexpTokenizer* from the nltk package allows the user to predefine what type of pattern it keeps for tokenization. In this case, the pattern is specified so that only words without numerical characters are kept. Moreover, the tokenizer removes interpunction in the different languages, such that even,

for example, Chinese punctuation is removed. Then, it lower cases everything. It is worth to mention that the *CountVectorizer* method that generates the unigram matrix already lowercases everything, so it is not necessary to lowercase beforehand. Next, the tokenizer returns the list of words. However, the pipeline expects as preprocessed data, one string of words or tokens seperated by whitespaces. Therefore, after any tokenization algorithm, the tokenize sentence function is used to join the list in to this format. The code corresponding to these steps is displayed below. It is expected that the coverage will improve, as the total number of different tokens decreases after this step. Also, the prediction capabilities should increase as uninformative tokens such as '.' are being deleted.

The coverage and weighted F1 score is given in Table /1. As expected, the coverage has significantly increased as well (about 11 %) and so has the predicting power (about 5 %).

```
def tokenize_sentence(sentence):
    tokenizer = nltk.tokenize.RegexpTokenizer(r'\b[^\d\W]+\b')
    sentence = tokenizer.tokenize(sentence)
    sentence = ' '.join(sentence)
    return sentence

def preprocess(sentence, labels):
    sentence = sentence.apply(tokenize_sentence)
    return sentence,labels
```

3.2 Stemming

Stemming is the process by which words are reduced to its roots and as such, the matrix of token counts might become less sparse. One of the most common stemming algorithm is the Porter algorithm. The issue with this algorithm is that it is build for stemming English words. However, because different European languages share similar word endings, it might have some success outside of English too. Nevertheless, other algorithms such as Snowball support more languages, but they need the language as input, which is not possible at this step of the pipeline. Thus, the Porter algorithm is applied, although it is not expected to significantly improve the results since languages are uniformly distributed through the data. As observed in Table 1, both coverage and F1 score stay the same as in the baseline case.

```
def tokenize_sentence(sentence):
    porter_stem = nltk.stem.PorterStemmer()
    sentence = ' '.join(porter_stem.stem(word) for word in
    sentence)
    return sentence
```

3.3 Sentence Splitting

Sentence splitting is of particular interest when the text is analyzed by its lexical, semantic, or even syntactic definitions. However, this is not the case. As such, the coverage is not expected to significantly improve, but the training data might increase in size, each containing shorter sentences, which can alter the prediction power. The results displayed in Table 1 show that the coverage stays the same, but the F1 score has, unexpectedly, decreased.

```
def preprocess(sentence, labels):
    df = pd.concat([sentence, labels], axis=1).reset_index()
    df['Text'] = df['Text'].apply(lambda x : nltk.tokenize.
    sent_tokenize(x))
    df = df.explode('Text')
    df = df.dropna()
    sentence = df.loc[:,'Text']; labels = df.loc[:,'language']
    return sentence, labels
```

3.4 Identifying Character Encoding

As mentioned in Section 2, the issue with word tokenization is that languages such as Chinese, Japanese, and Korean do not get properly split. In these languages, each character represents a word, while a set of characters describe a sentence. One way to properly tokenize them without using the label is by looking at the alphabet used in a sentence. Now, if the majority of a sentence is written with logograms, its tokenization will instead be executed at "character" granularity (although it is a word per se). In order to do this, the *unicodedata* package is used, which distinguishes some alphabets by its computer incoding, not using a dictionary. This means that these features are extracted from any sentence and thus not language specific techniques. The results depicted in Table 1 show a significant coverage and prediction power improvement. The former is not as great as with the punctuation removal, but the latter yields a better score.

```
import unicodedata as ud

def contains_cjk(unistr):
    for uchr in unistr:
        try: ud_name = ud.name(uchr)
        except: ud_name = ''
        if any(x in ud_name for x in ['CJK', 'HANGUL']):
            return True
    return False

def contains_latin(unistr):
    for uchr in unistr:
        if any(x in ud.name(uchr) for x in ['LATIN']):
            return True
    return False

```

```
def alphabet_is_cjk(unistr):
    cjk_chr = 0; other_chr = 0
    idx_list = []
    for idx, uchr in enumerate(unistr):
        try: ud_name = ud.name(uchr)
        except: ud_name = ''
        if any(x in ud_name for x in ['CJK', 'HANGUL']):
            cjk_chr += 1
            idx_list.append(idx)
        else:
            other_chr += 1
    cjk_prop = cjk_chr/(other_chr+cjk_chr)
    if cjk_prop >= 0.5:
        return True, idx_list
    return False, idx_list

def alphabet_is_latin(unistr):
    latin_chr = 0; other_chr = 0
    idx_list = []
    for idx, uchr in enumerate(unistr):
        try: ud_name = ud.name(uchr)
        except: ud_name = ''
        if any(x in ud_name for x in ['LATIN']):
            latin_chr += 1
            idx_list.append(idx)
        elif uchr == ' ':
            idx_list.append(idx)
        else:
            other_chr += 1
    latin_prop = latin_chr/(other_chr+latin_chr)
    if latin_prop >= 0.5:
        return True, idx_list
    return False, idx_list
```

```
def tokenize_sentence(sentence):
    if contains_cjk(sentence):
        alph_is_cjk, idxs = alphabet_is_cjk(sentence)
        if alph_is_cjk:
            final_sentence = ''
            for idx in idxs:
                final_sentence += sentence[idx] + ' '
            return final_sentence

    elif contains_latin(sentence):
        alph_is_latin, idxs = alphabet_is_latin(sentence)
        if alph_is_latin:
            final_sentence = ''
            for idx in idxs:
                final_sentence += sentence[idx]
            sentence = final_sentence

    tokenizer = nltk.tokenize.RegexpTokenizer(r'\b[^\d\W]+\b')
    sentence = tokenizer.tokenize(sentence)
    return ' '.join(sentence)
```

3.5 Complete Case

Finally, the two seemingly best options (punctuation removal and encoding) are combined. The results in Table 1 show a great improvement in coverage as well as in predictive capabilities. Compared to the baseline, they have increased by about 17 % and about 12 %, respectively. This might be due to the fact that only relevant tokens are taken into account at the same time that the tokenization is properly done across languages with different alphabets. The confusion matrix can be depicted below. It can be observed how the languages do not get confused, as all of them are properly described by the matrix of tokens.

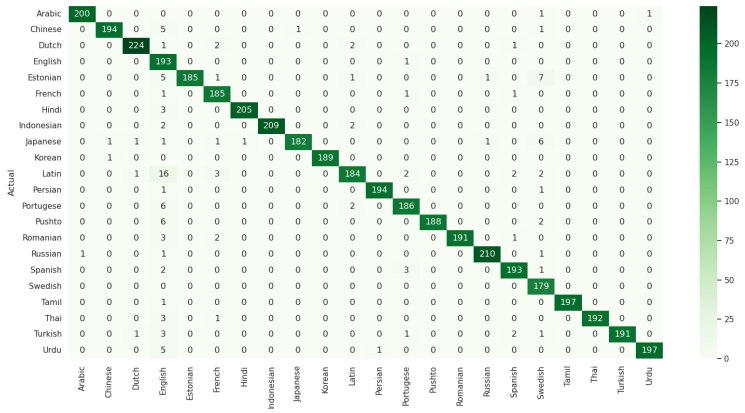

Figure 3.2: Confusion matrix of the preprocessed classification with 1000 words and the Naive Bayes classifier.

	Coverage	**Weighted F1 Score**
Baseline	0,3323	0,8513
Removed Punctuations	0,4474	0,8994
Stemming	0,3323	0,8513
Sentence Splitting	0,3328	0,8506
Character Encoding	0,4268	0,9131
Complete*	0,5002	0,9709

Table 1: Results obtained after applying different preprocessing steps to the corpus of 1000 words and using the Naive Bayes Classifier. *The complete case only consists of removed punctuation and character encoding.

4 Classifiers

Different classifiers were tested on both the baseline case, i.e. without any preprocessing, and on the initially best performing preprocessing method, tested on the Naive Bayes Classifier, which is a combination of character encoding and tokenization. Both cases consider word granularity with a vocabulary size of 1000 words. The running time of each algorithm is also included. All results are depicted in Table 2.

4.1 Naive Bayes

```
clf = MultinomialNB()
```

The initial classifier is the Naive Bayes classifier, which bases its prediction on Bayes rule [2]. In the base case the algorithm yields and F1-score of 0.851, while in the preprocessed case that goes up to 0.971 and takes 0.39 seconds to do so. This is a good and fast result.

4.2 Linear Discriminant Analysis

```
clf = LDA()
```

The following classifier is a Linear Discriminant Analysis, an algorithm that maximizes separability eventually also using Bayes rule [1]. It takes 5.08 seconds for an F1-score of 0.902 in the base case and 0.972 in the preprocessed case. A slight improvement compared to the Naive Bayes classifier with the cost of a longer execution time.

4.3 Support Vector Machine

```
clf = LinearSVC()
```

Next, a Support Vector Machine with a linear kernel is applied. A classifier using kernels to maximize the hyperplanes separating different languages from each other [4]. It takes 0.73 seconds for an F1-score of 0.905 in the base case and 0.976 in the preprocessed case. This can easily be considered the best performing classifier as it is both very fast and yields the highest F1 score.

4.4 Neural Network

```
clf = MLPClassifier()
```

Then, a simple and unoptimized neural network algorithm is applied by using the MLP classifier. Deep learning trains connected perceptron layers minimizing a log-loss function [3]. It takes 46.01 seconds for an F1-score of 0.901 in the base case and 0.972 in the preprocessed case. These results are not

	Baseline F1 score	Prep. F1 score	Time (s)
NaiveBayes	0,8513	0,9709	0,3930
LDA	0,9016	0,9722	5,0845
SVM	0,9047	0,9759	0,7340
NeuralNetwork	0,9005	0,9719	46,0139
LinearRegression	0,9025	0,9737	5,0101

Table 2: Results obtained after applying different classification algorithms to the corpus of 1000 words while using the best preprocessing steps.

considered good as it takes a long time and yields an average F1-score. However, as deep learning models often benefit greatly from hyperoptimization, these results are not to be taken to seriously.

4.5 Linear Regression

```
clf = MLPClassifier()
```

Finally, a simple linear regression is applied. It takes 5.01 seconds for an F1-score of 0.903 in the base case and 0.974 in the preprocessed case. These results are also good and fast, although the SVM with a linear kernel does a better job on both aspects.

5 Conclusions

Throughout the project, the different mechanisms influencing a language recognition pipeline have been explored, adapted and analyzed. It is found that a lot of factors play a role: word/character granularity, vocabulary size, preprocessing steps, corpus data quality, classifier used, etc. In the case of word granularity, the pipeline mostly struggled with recognizing Chinese and Japanese. This problem has been resolved in preprocessing by using a character encoding technique followed by proper tokenization. Finally, although most classifier alogorithms perform similarly, the Support Vector Machine is found to deliver the best combination of performance and speed. Although not the inital objective, combining all best practices, an excellent F1 score is obtained, even more when considering the word granularity and small vocabulary size of 1000.

References

[1] Sklearn linear discriminant analysis. https://scikit-learn.org/stable/modules/classes.html#module-sklearn.discriminant_analysis.

[2] Sklearn naive bayes. https://scikit-learn.org/stable/modules/naive_bayes.html.

[3] Sklearn mlp classifier. https://scikit-learn.org/stable/modules/generated/sklearn.neural_network.MLPClassifier.html?highlight=mlp#sklearn.neural_network.MLPClassifier.

[4] Sklearn support vector machine. https://scikit-learn.org/stable/auto_examples/svm/plot_separating_hyperplane.html?highlight=support%20vector%20machine.