Word Level Language Identification in Code Switched German-English Text

Andrew Bell

Bachelor of Science in Computer Science
The University of Bath
2021/2022

Word Level Language Identification in Code Switched German-English Text

Submitted by: Andrew Bell

Copyright

Declaration

This dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Bachelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university or institution of learning. Except where specifically acknowledged, it is the work of the author.

Abstract

Automatic language identification is a well studied task, but is not effective on code switched text, where multiple languages are present. Instead, we must identify each word separately, which is known as word level identification. This is important because other natural language processing tasks suffer in performance when encountering code-switched text, and so a system to identify which words belong to which language is necessary.

In this project, we gather code-switched text, containing German and English, from Twitter, popular social media platform (Twitter). We implement a character n-gram based model, and three different classification procedures. Existing implementations of various machine learning classifiers were utilised and tested with each procedure. The best F1-score achieved was 90.8%, which is 23.7% higher than a naive n-gram classifier and 50.4% higher than a basic dictionary approach. To our knowledge, this implementation is the first word-level language identification system for German-English code switched text.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, Dr. Kochmar for her valuable insights and help throughout the project, and my girlfriend Sanugi for her crucial help with proofreading, support and motivation.

Chapter 1

Introduction

The problem of automatic language identification (LID) is in the field of natural language processing (NLP) and has been solved successfully in single language texts. This type of LID is referred to as conventional or grouped language identification. A group of words (such as a sentence) is tagged with a single language. With conventional identification, texts can be classified between languages with up to 100% accuracy rate simply using dictionaries and word n-grams.

However, much of the text used for NLP tasks comes from social media, which tends to be noisy. This noise includes misspellings, non-formal writing and code switching (CS). CS is the use of more than one language in a single instance of communication. When considering a text with CS, the problem becomes more complex. Each word must be identified separately (or else no information can be gleamed). This is known as word level identification [11], and is necessary for data that contains code switching.

The task of language identification for code-switched text is important because other NLP tasks often assume their data contains only one language, and suffer in performance when they encounter text of this kind [7]. This means we need a way of dealing with code switched text when it is found. Language identification on a word level allows us to either append a language tag to each word, for the next process to address, or translate all words into a single language as a form of normalisation.

This project addresses the problem of automatic word level language identification in code-switched social media text. The process of translation into single language was not tackled in this work. German and English were the languages of choice, which unlike other pairs, have not been studied for this task. Language pairs such as English and Hindi have been solved very successfully, but German and English are unique as they are very close, in alphabet, lexicon and grammar.

Furthermore, cultural and historical factors are explored in Section 2.2 which contribute to the difficulty and importance of automatic LID between these two languages. This includes the language origins of English, the modern influence between the languages and the high proportion of native German-speakers who can also speak English.

Chapter 2

Literature and Technology Survey

2.1 Code Switching

Code switching[1] is the use of more than one language in a single instance of communication [18]. The most common forms of CS are inter-sentential and intra-sentential. The former is when language changes between sentences, and the latter is when it changes within a sentence. The term code mixing is used either as a synonym for code switching, or to refer specifically to intra-sentential CS. In this project, intra-word code switching is not considered, and so we assume each word has exactly one language. Intra-word code switching is found less frequently, and so it is less important to consider.

.

Example 1 *Intra-sentential code switching*

"Wie soll ich denn meine träume manifestieren wenn ich gleichzeitig nach dem low expectations, low disappointment prinzip leben will?"

(How am I supposed to make my dreams come true when I also want to live by the principle of "low expectations, low disappointment"?)

Example 2 *Inter-sentential code switching*

"Warum hast du das gesagt? That really hurt my feelings."

(Why did you say that? That really hurt my feelings.)

There are a number of reasons why CS occurs. One of the main social theories states that it is used within communities to establish membership. There are many bilingual or multilingual communities and the use of a particular kind of CS shows belonging in a group [11, 12]. This type of CS is present especially on social media, where a single post can be read by a huge number of people, and thus demonstrating membership to a community is more desirable. This use of CS can be purposeful but more often happens subconsciously depending on the intended recipient(s) of the utterance or message is.

There are also linguistic purposes for CS. In particular, when a certain meaning is desired, but a single language may not have the lexicon to express this meaning. This kind of CS is more

[1] Also referred to as codeswitching or code-switching

common in speech, where a speaker has limited time to think about what they will say, but it also occurs in textual forms. Social media in particular is a rich source of code switched text, as posts generally have a more colloquial tone, and people write in any manner they are comfortable.

Code switching has been observed in texts as early as the medieval period in England [33]. This concept is certainly not new, but early researchers believed it occurred due to inadequate language skills. However, in recent works, the opposite has been observed. Speakers who code switch tend to have a good understanding of the languages they are utilising [15]. This is because CS is not just any word replacement, but rather a purposeful and grammatically sound mix of multiple languages. There are clear examples of mixed language that bilingual speakers would identify as nonsense, and these we would not consider to be code switching [18].

However, the rules that govern code switching are highly debated. A minimalist theory states that the constraints of CS are at most the union of the grammars of each language [9]. This conclusion is reached because for every attempt of formalising the mediation between grammars, there are significant exceptions when applied to naturalistic code switching [18]. The implication of this is that CS is not well-defined and therefore classification as such is not objective. One person may identify a text as CS, where another may identify it as nonsense, even with a similar multilingual knowledge and ability.

Although it may be hard to identify whether a sentence as a whole is considered CS, labelling the language of each word in a sentence tends to be far more objective. Given a code switched text, people with the appropriate skills tend to identify this with ease, and the results between annotators usually concur. This is demonstrated in the work of Nguyen et al. who calculated an inter-annotator agreement value of 0.935 and Mave et al. who calculated a Cohen's kappa value (Equation 4.1) of 0.98 for word level language annotation [26, 21]. There can still be a level of ambiguity as it is not always clear whether a word belongs to one language or another - words can be borrowed and shared between languages. To deal with this, annotators are often given an ambiguous or 'both' option. This incites more consistency in word level labelling.

2.2 Geman-English Code Switching

English is primarily considered a Germanic language and many of its words have German origin. Additionally there is significant modern influence between the two languages, due to a variety of factors including immigration between German and English speaking countries but also historical factors such as the occupation of Germany following WW2. Although influence between languages is common, there is a unique similarity between English and German. This can be seen through the bidirectional interchange of words, in particular, in German, many English words are used for scientific and technological concepts such as "Software" or "Computer". Additionally there are German words used in the English language such as "abseil" or "kindergarten". These influences and word borrowings add to the already prevalent ambiguity of differentiation between German and English words.

When it comes to CS, it is most commonly found within bilingual communities of English and German-speakers, much like other language combinations. However, there is also currently many German-speakers that despite not being part of a bilingual community, use English words in place of German ones during speech or writing. This is known as Denglisch (Deutsch-English) and is fairly controversial within German-speaking countries, due to a reported negative impact on the conservation of the German language [17]. Denglisch can include word borrowing; many words from the English language are borrowed, as an equivalent does not exist. English phrases and words can become popular because of cultural, social or economic influence. However, these cases tend to be more accepted in German-speaking countries, whereas the term Denglisch usually refers to the use of English words *instead* of the existing German word. Denglisch may also include adaptations or Anglicisms of German words, which is one of the main points of criticism against Denglisch. It is also worth noting that Denglisch is enabled by the fact that German-speaking countries tend to teach English from a young age. This means that even if a native German speaker is not bilingual, they tend to have a good knowledge of the English language [3].

To conclude, it is important to understand that text containing both English and German words can appear for a number of reasons, including CS, Denglisch and word borrowing. However despite the varying reasons, language annotation on a word level need not discriminate between them. There is a significant level of ambiguity in identification that is unique for German-English code switched text, especially due to word borrowing and the Germanic origins of the English language. It is harder to distinguish a German word from English than it is for two less similar languages. This makes automatic language identification a more challenging task, especially on a word level.

2.3 Natural Language Processing with Code Switched Text

Natural Language Processing tasks such as normalisation, machine translation and parsing suffer in performance when encountering code switched data [7]. Normalisation in this context refers to standardising a text into a certain form (that is easier to process) and parsing is the process of analysing the syntax of a sentence. These tasks in particular are reliant on language conformity, and therefore CS can cause problems. A solution to this is identifying the language on a word level, as a first step, before other techniques are employed. After identification, words can be translated as to produce a text containing a single language [35]. This process can be thought of as a type of normalisation, as it translates the text into a single language. In this project, the normalisation step is not implemented, but as an effective LID implementation, it could be utilised in a wide variety of NLP tasks that rely on the text being a single language or having token level language tags.

NLP tasks have also been implemented directly from code switched data, without the need for LID. This includes problems such as offensive text identification [20], sarcasm or hate speech detection [27], language models for speech recognition [39], sentiment analysis [28] and machine translation [40]. These systems are specialised to perform with code switched text, though this has the disadvantage of not working on single language data as efficiently as one designed to do so. Therefore to create more general solutions to code switched NLP problems, it is preferable to process code switched text into an appropriate form using LID before use in other NLP systems. However, if performance is desired most, then a specialised solution for CS text is more appropriate.

2.3.1 Data and Corpora

One must have access to a large amount of data when processing code switched text, which is often retrieved from public corpora or shared task data sets. However, besides these, a proven approach to finding code switched text is to use social media with language and geographical filters to find users that tend to code switch in their posts. These users' posts can then be accumulated to create a corpus of code switched text [19]. When dealing with social media, data tends to be noisy. Nguyen et al. ignore noise in the form of numeric tokens, forum tags, links, proper nouns, usernames and 'chat words' using regular expressions [26]. Since many works that process code switched text gather data from social media, it is common for this data to be processed before use.

If no data can be found (either from a public corpus or social media), it is possible to use artificially generated code switched text. Sanad et al. present a tool which can generate code switched sentences [31], which is expanded and utilised by Tarunesh et al. to generate Hindi-English code switched text from Hindi monolingual text. This is shown to have performance on par with natural code switched text from human evaluation [37]. This tool is useful in any NLP task where code switched data is required, but scarcely found.

2.3.2 Evaluating Performance

Typically, machine learning systems are evaluated by predicting labels for the test set, and comparing these to the true labels. Then, certain metrics are calculated to compare performances with a baseline or previous work. Accuracy is the most common metric found and is easy to understand, but should not be relied upon. Nguyen et al. suggest that further metrics should be used [26], as accuracy can be misleading. For example, if 90% of the test set belongs to the "negative" class, then a classifier that picks negative every time will achieve an accuracy of 90%. To deal with the problem of an imbalanced data set, we need to use further metrics: precision, recall and F1-score. In the same example, the classifier would receive a perfect recall, but poor precision score. By using the F1-score[2], which is the harmonic mean of precision and recall, we are able to represent both equally [32].

However, in some cases the F1-score can be flawed when precision is more important than recall, or vice versa. For instance, in medical diagnoses, false negatives can be worse than false positives. Therefore, we must analyse which of these are more important in task at hand (see Section 8.1).

To calculate these metrics, first a confusion matrix is produced. Example 3 demonstrates a matrix with two classes (Positive and Negative).

Example 3 *2x2 Confusion Matrix*

		Predicted	
		Positive	*Negative*
True	*Positive*	*tp*	*fn*
	Negative	*fp*	*tn*

A 2x2 confusion matrix demonstrates the number of true positives/negatives (tp/tn) and false positives/negatives (fp/fn). These values can be used to calculate performance metrics [14]

[2] Also known as f-score or f-measure

with Equations 2.1 - 2.4.

$$Accuracy = \frac{tp + tn}{tp + fp + tn + fn} \tag{2.1}$$

$$Precision = \frac{tp}{tp + fp} \tag{2.2}$$

$$Recall = \frac{tp}{tp + fn} \tag{2.3}$$

$$F1\text{-}Score = 2 \cdot \frac{precision \cdot recall}{precision + recall} \tag{2.4}$$

In the case that there are more than two classes, metrics are calculated on a per class basis. For language identification, this could mean there are three languages and each one would have its own precision, recall and F1-score. These can be averaged to find overall scores.

The performance metrics can be compared to a baseline to indicate either success or failure of a system. Many works compare these to the current state of the art, but it is often the case that the exact problem has not been addressed. In this case, the next best solution can be compared - for example in the case of code switched NLP, the performance can be compared to that of a system designed for single language texts but applied on code switched data. However, since this has already been shown to be ineffective [7], we compared our system to a basic dictionary approach and a naive character n-gram classifier.

2.4 Language Identification

2.4.1 Grouped (Conventional) Identification

Language identification is an area of NLP that has been tackled extremely successfully in single language texts. It is employed by systems such as Google Translate[3] in order to identify the language entered by a user before applying machine translation. Machine learning algorithms have been able to identify the language of input texts with up to 100% accuracy in European test sets [23].

Automatic LID can be done using a variety of different methods. A simple approach is to use a dictionary lookup to find the likelihood of each word belonging to a certain language and then classify the text as the most frequently appearing language of the word. This technique can be employed at the word level or higher (as individual characters do not appear in dictionaries).

Despite this approach being applicable to conventional LID for a group of text and word level code switched text identification, it is unsuitable on its own for either task. This is because in real texts, inconsistencies such as spelling errors, alternative spellings, region/field specific terminology and colloquial terms are found. A dictionary approach is also resource intensive and not scalable; while a dictionary has to become larger to identify more words, other models can remain the same size. Finally, this approach is unable to take any context into account, leaving each word on its own for identification.

[3] translate.google.com

However, the dictionary approach can be extremely effective when paired with modelling techniques such as word n-grams, with accuracies of up to 100% being achieved [41] for conventional LID. Word n-grams model the frequency of word sets of length n (Example 4).

Example 4 *Word bi-grams (n=2)*

$$\text{''}\textit{This is a sentence}\text{''} \xrightarrow[\textit{bi-grams}]{\textit{word}} [\text{''}\textit{This is}\text{''}], [\text{''}\textit{is a}\text{''}], [\text{''}\textit{a sentence}\text{''}]$$

The frequency of n-grams can be recorded in a data structure, and when a word is encountered, the frequency of all its n-grams are summed. This gives a score that can be used either naively to predict a language, or used as a feature within a machine learning classifier.

When it comes to machine learning approaches, there are a number of options to address the problem of LID. First a modelling approach is selected, for example word n-grams. Then a classification method is employed, of which there are many possibilities. Support Vector Machines (SVM) and Multilayer Perceptron (MLP) are used very successfully for this task [36]. SVM is a kernel based classifier that attempts to find the hyperplane that best divides groups and MLP is a neural network based classifier. In the task of conventional language identification, it was found that SVM outperformed MLP [13].

2.4.2 Word Level Identification

The problem of LID becomes more complicated with code switched text. CS can occur on different levels, and conventional language identifiers lose accuracy when faced with code switched text [7]. This approach, which does not consider the language of each word separately is ineffective for mixed language text. The main solution to this, is to identify language on a word level, rather than as a group of text.

Example 5 *Word level language identification*

$$\text{“}\textit{mein leben ein disaster}\text{”} \Longrightarrow [0, 0, 0, 1]$$

(0 = German and 1 = English)

On a word level, a more concise approach is needed to achieve accurate results. This is because identifying the language a group of text allows the use of more information and has a higher tolerance to error. In spite of this, there have been very successful solutions to this problem. The difficulty is, one cannot collate together any words for identification. This makes word n-grams far less useful, as we must only use n=1, which is essentially a dictionary classifier. Dictionary classifiers can be very successful in formal texts, but with social media texts, a dictionary only approach is insufficient for this task. Instead, the basis for many methods is character n-grams.

Example 6 *Character tri-grams (n=3)*

$$\text{''}\textit{sentence}\text{''} \xrightarrow[\textit{tri-grams}]{\textit{character}} [\text{''}\textit{sen}\text{''}], [\text{''}\textit{ent}\text{''}], [\text{''}\textit{nte}\text{''}], [\text{''}\textit{ten}\text{''}], [\text{''}\textit{enc}\text{''}], [\text{''}\textit{nce}\text{''}]$$

These character n-grams can be used in frequency tables in the same fashion as word n-grams. They are used along with other statistical methods to model textual data, and a machine learning classifier can be utilised to achieve high levels of accuracy.

To identify language on the word level for code switched Turkish-Dutch text, Nguyen et al. trained a system using a multitude of classifiers, the best accuracy achieved being 98% [26]. This is achieved through the use of Conditional Random Fields (CRF) which takes into account the word itself and context. These are modelled by a joint dictionary and character n-gram approach of maximum length 5. This is a supervised approach, so the data was annotated and a smaller dataset was used to train the classifier. The dictionaries are generated by tokenised web pages written in the two respective languages. It is clear this is an extremely successful solution and it gives merit to this method.

Voss et al. produced a classifier to identify language on the word level for code switched text including romanised[4] Moroccan Arabic (Darija), French and English [38]. Using Latent Direchlet Allocation (LDA) to cluster the data, they achieved accuracy scores of 93.2%, 83.2% and 90.1% for Darija, English and French respectively. This approach is unsupervised and was chosen due to a lack of data sets for Darija. There are a number of reasons the classifier achieves lower accuracies than that of Nguyen et al. More languages were recognised, an unsupervised classifier was used, a contextless approach was taken and there was ambiguity in the transliteration. Since Darija uses an Arabic script, users transliterate the words into the Latin alphabet in order to write it on social media, and not every speaker of Darija will choose to write these the same way. This ambiguity was not present in the data used by Nguyen et al. as Turkish and Dutch share the same alphabet.

However, Mave et al. were able to reach an accuracy score of 96% on Hindi-English code switched text, a combination of two languages that do not use the same script. Their approach also involves CRF and the context of each word is taken into account for its identification. This approach is also supervised, but the better accuracy scores than those of Voss et al. could be attributed to the abundance and conformity of data. Hindi-English code switching is extremely common on the internet and there are an estimated 528 million Hindi speakers as of 2011 [4]. This certainly reduces the ambiguity of language transliteration as there are stronger romanising conventions.

Other works in this field include MSA-Egyptian and Spanish-English identifiers using a Recurrent Neural Network (RNN) approach [30]. Hindi-English / Spanish-English identifiers were tested using both CRF and RNN and the CRF model had the higher performance by a significant margin [21]. Additionally, a Sinhala-English identifier was trained using a variety of classifiers, the best performing being Random Forest [34].

In conclusion, the best performing systems tend to use a model that includes a dictionary and character n-gram element. Additionally, the successful approaches take context into account and usually make use of a sophisticated supervised machine learning classifier, such as SVM or MLP. There are currently no published works on a code switched German-English word level classifier, which is the focus of this project.

[4] Transliterated into the Latin alphabet by the original authors

Chapter 3

Method Outline

The goal of this project is to implement a word level language identification tool for code-switched German-English social media text. The following outline shows our approach to complete this task:

Data

1. Collect data by finding code-switching users on social media, and collate their posts to form a dataset. This dataset is two-dimensional: A list of posts, each post is a list of words
2. Clean and annotate the dataset; each word has its own language label
3. Flatten the data into a one-dimensional list of words. To preserve the structure of the posts, pair each word with its corresponding position value in the post
4. Split data into training and test set

Model

5. From the training set and common word lists, create a *model* in the form of two frequency tables, one for each language.
6. For both the training and test set, use the *model* and further data to extract features for each word.

Classify

7. Implement classification procedures and import machine learning classifiers
8. Use training features and labels to train each imported classifier with each procedure
9. Use each classifier/procedure combination to predict labels from the test features
10. Compare the predicted labels to true test labels and evaluate performances

Chapter 4

Data

4.1 Collection

There were a number of options to collect data for this task. The first and easiest would be a public corpus or shared task dataset. However for this particular language pair, there exists (to our knowledge) only one such dataset [5] and the source of its data are Wikipedia articles, which are highly formal. For this reason, it is not appropriate for the purpose of this task, which is to be able to identify language on a word level for code-switched *social media* text.

With this in mind, we looked towards the online social media platform, Twitter[1]. It is a vast source of social media text, as all of its posts (tweets) are freely and publicly available. Additionally, Twitter has an API which allows us to search for tweets by a number of parameters. Our approach to finding data was inspired by that of Maharjan et al. [19]. We downloaded a section of the monthly German tweet dataset [16], and filtered them to identify those that contained at least one English word. Filtering was performed automatically, by first removing any common words (between English and German) from a list of English words. This list was then used to filter the tweets to find ones containing English. Although this is a form of LID, it is a basic dictionary approach which was used detect the presence of code switching, not identify every word.

The filtered tweets were compiled together, and the ids of the users who posted them were recorded. This formed a list of users who were likely to code switch. Using Twitter's API, we collected all the tweets that these users had posted. The reasoning is, users who post tweets containing both English and German words, are more likely to use CS. The result of this step was a shorter list of tweets, containing all the tweets of the users who published those filtered tweets (except for those in the filtered list to begin with). This formed our raw dataset. It was more limited than expected because despite using recent tweets, most of the users who had published them were either deleted accounts or inaccessible (indicated by errors thrown by Twitter API). The end result was 220 tweets, made up of 2984 words in total.

[1] twitter.com

4.2 Cleaning

The cleaning process for this data was fairly simple. Due to the nature this work, the data does not have to be normalised or cleaned thoroughly. The data was cleaned by removing any unnecessary white space, links, hashtags and other irrelevant symbols.

The basis for this approach uses character n-grams, and our implementation of them ignores any non-alphanumeric characters. So by keeping these in the dataset, it pushes the responsibility of cleaning to the modelling stage, allowing us to have more control when it comes to extracting the features. Were we to clean everything at this stage, information that may be helpful in the modelling stage could be lost.

4.3 Annotation

The data was annotated manually, with three possible labels, shown in Table 4.1. Because the data in question is noisy and contains many unusual symbols, and words from other languages, we included a third language class. As explained in the annotation instructions (Appendix B), "Other" refers to any words that are neither German nor English.

Table 4.1: Labels used for annotation

Labels	Language
0	German
1	English
2	Other

The data was annotated in its entirety by the author, who is fluent in English and German. Annotations were made on a total of 2984 words. This was performed using a custom script which displays a sentence and takes an input to label each word. This produced an output file of all the tweets in one column and the labels in the other. Annotations were made firstly according to the word itself, and if the annotator is unsure whether this word is German or English, it is decided from context. Finally, if it is neither German nor English it should be labelled as other.

To ensure the accuracy of these annotations, a second bilingual annotator was asked to annotate a random sample of the data. 10% was randomly selected and sent along with annotation instructions (Appendix B) to the second annotator. These annotations were compared with our annotations of the same data, and a Cohen's Kappa (Equation 4.1) of 0.795 was calculated. This statistic is a measure of inter annotator agreement, and takes into account the hypothetical chance of agreement, making it a more reliable measurement. By medical standards, a Cohen's Kappa of 0.8 is classed as a strong agreement level [22].

$$k = \frac{p_0 - p_e}{1 - p_e} \tag{4.1}$$

Where p_0 is the relative agreement (analogous to accuracy);
p_e is the hypothetical chance of agreement.

4.4 Pre-Processing

In order to process the data into an appropriate form for modelling and classification, the data had to be formed into a one-dimensional list of words from a two-dimensional list of tweets. The reason for this is that we utilise the scikit-learn[2] library for classification, which does not support nested lists unless each sublist is of identical size. To comply with this format, a flattening function was utilised. However, this alone would lose the structure of the tweets, so to preserve this information, each word was paired with its corresponding position.

Example 7 *Flattening*

[[a,b,c,d], [e,f,g]] $\longrightarrow$ *[(a,0),(b,1),(c,2),(d,3),(e,0),(f,1),(g,2)]*

Although the resulting list is still two-dimensional, the sublists are all the same size. This is shown in Example 7, where a list contains two sublists. The first sublist has four elements and the second has three. The result of the flattening results in a list with 7 (4+3) sublists, but they are all of size two. This allows it to conform with the scikit-learn library. The label data was also flattened, but without any corresponding positions.

The final step of data processing was splitting the data into training and test sets, using the scikit-learn library. The annotated data was split into a ratio of 0.67:0.33 (training:test). Since the data was more limited than expected, we used a significant portion of it to test, which helped add more evidence to the experiment's validity.

4.5 Data Summary

The number of words in each set, and language class can be seen in Table 4.2. German was the dominant language in the set. The reason for this is likely how the data was collected. Users were found using tweets tagged as German to begin with, meaning there is a high likelihood of tweets containing more German than English. Furthermore, German-English code switching appears altogether more commonly in the form of mostly German text with a few (2-4) English words present. This was confirmed through tests with other tweet datasets. It was found that tweets tagged with English as the language didn't contain nearly as many German words as vice versa.

Table 4.2: Number of words in each language class in dataset

Language	Words			
	Total	Proportion	Training set	Test set
Total	2984	100%	1999	985
German	2489	83.4%	1665	824
English	310	10.3%	203	107
Other	185	6.2%	131	54

The reason for this may be that native German-speakers, more often than not, have an understanding of English, but the inverse is not true. This is indicated by census data: In Germany, 56% [3] of the population are able to speak English. On the other hand, there are proportionally very few German-speakers in English-speaking countries such as the United

[2] scikit-learn.org

Kingdom or United States. This bias in the dataset must be considered in classifier choice, as certain classifiers are able to take this into account better than others.

Chapter 5

Modelling

The purpose of modelling is to extract meaningful features from the annotated textual data. Training a classifier relies on a set of features for each word that contain numerical/binary data only. The most important feature that we extracted were character n-gram frequency scores. To calculate these, we created a frequency table for each language. This model, along with word lists and textual information were used to extract the features.

The modelling approach was mostly programmed from scratch. The reasoning was that the modelling needs to suit the task and data much more closely than the classifier.

5.1 Model Creation

The basis of the model is character n-grams. These were extracted from each word using function 5.1. It takes a word and integer n as input, and returns the character n-grams of this word. It works by taking each letter of the word, and makes a string of it and the next $(n-1)$ characters, unless the end of the word is reached, in which case the string is discarded. The collection of all non-discarded strings from a word are the character n-grams. Additionally, non alphanumerical characters are ignored, which in this implementation means that any n-gram that contains one is discarded.

Function 5.1: Character n-grams

```
def character_ngrams(word,n):
    word = word.strip().lower()
    result = []
    i=0
    for i in range (0,len(word)):
        ngram = ""
        append = True
        for j in range (0,n):
            if (i+j>=len(word) or not word[i+j].isalpha()):
                append = False
                break;
            ngram+=word[i+j]
        if (append):
            result.append(ngram)
    return result
```

To create the model, two dictionaries (hash tables) were defined and each populated using function 5.2. It works by first using function 5.1 to extract the character n-grams of each word in the data and then incrementing each n-gram's position in the hash table. The value is increased by a predefined weight in the argument. If no n-grams are found, it uses $(n-1)$-grams until at least one is found. This ensures that words with lengths smaller than n are still represented.

Function 5.2: n-gram Frequency Table

```
def ngram_table(dict, total, data, n, weight):
    for word in data:
        ngrams = character_ngrams(word, n)
        i = n-1
        while (len(ngrams)==0 and i>0):
            ngrams=character_ngrams(word,i)
            i = i-1
        for ng in ngrams:
            total+=1
            if ng in dict:
                dict[ng]+=weight
            else:
                dict[ng]=weight
    return dict, total
```

The total number of n-grams were recorded in this process and every value in each hash table was divided by the corresponding total. This normalisation step is important because a different number of words for German and English would result in biased hash tables. The 'other' label does not have its own frequency table as there is not enough data for it, nor does it constitute a formal language.

The data used to create these frequency tables was initially just the training data, but it was found that the tables could be improved by processing a word list for each language. These word lists contained 1000 frequently used words in German and English respectively [2, 1]. The weight parameter is used to put more emphasis on the n-grams produced by the training data, than the n-grams from the common word lists. By creating frequency tables for each language using both common word lists and the training data, we were able to make well-rounded frequency tables, that are not entirely specific to the training data.

5.2 Feature Extraction

5.2.1 Character n-gram Frequency Scores (Numerical)

Using the model, frequency scores were calculated independently for each language. This means each word has a German and English frequency score. The frequency scores of a word were calculated using Algorithm 1.

Algorithm 1 *Character n-gram Frequency Score*

- *Extract the word's character n-grams, and for each n-gram, look up the value in its position in the given frequency table.*

```
• The sum of each n-gram's frequency score is the score for the word
```

This results in a numerical value, that represents a word's "score" for a particular language. The minimal value for this feature is 0, indicating that no n-grams were matched.

The frequency scores were the most important features. To demonstrate this, a naive classifier (Algorithm 2) was implemented .

Algorithm 2 *Naive Frequency Classifier*

```
• Fix n=3 (tri-grams)
• For each word in the data, calculate its frequency score for English and German.
    – If scores are both 0, choose 2
    – Else if scores are the same, return 0
    – Else if German score is higher, return 0
    – Else return 1
```

This naive classifier was able to achieve a maximum accuracy score of 68%. This showed that this feature is highly valuable.

5.2.2 Dictionary Scores (Binary)

Dictionary scores were found using the same lists of common words used to generate the character n-gram frequency tables. This means that essentially this feature is word 1-grams. Since the words in the lists were unique, the score is either 0 or 1, indicating the absence or presence of the given word in the list. This feature was tested with a basic classifier (Algorithm 3)

Algorithm 3 *Basic Dictionary Classifier*

```
• For each word, check its presence in the German and English word lists
    – If it exists in the German list return 0
    – Else if it exists in English list, return 1
    – Else return 2
```

This classifier achieved an accuracy of 51%. This score indicates that is a useful feature, but clearly a dictionary approach alone is unsatisfactory. The score could be improved by using larger word lists, but there are diminishing returns.

5.2.3 Position (Numerical)

This feature is extracted directly from the processed word data, which contains the word itself and the position value. Since the data had to be processed into a single dimension, it is necessary to include a position value to inform the classifier of textual structure. This feature is useful because from our observation, English words tend to occur toward the end of tweets. This feature is a numerical value ranging from 0 to the maximum possible number of words.

5.2.4 Other Features

Further textual features were extracted without the need for additional processing.

- *Length* (Numerical)
- *Is Uppercase* (Binary)
- *Is Capitalised* (Binary)
- *Begins with Vowel* (Binary)
- *Ends with Vowel* (Binary)
- *Contains Apostrophe* (Binary)
- *Contains Umlaut* (Binary)

Chapter 6

Classification

6.1 Procedures

Three different classification procedures were developed for this task; each one has:

Input: (`X_train`, `X_test`, `Y_train`, `clf`)
Output: `Y_pred`, the predicted values for `X_test`.

The `clf` object represents a classifier that is imported and passed into the function for use. These procedures do not implement a classifier, rather they use one in three different ways. For a description of the imported classifiers, see Section 6.2.

6.1.1 Standard

For the standard procedure, a given classifier is trained on the featurised training data. This classifier is used to predict labels from the featurised test data. This procedure is shown in Figure 6.1.

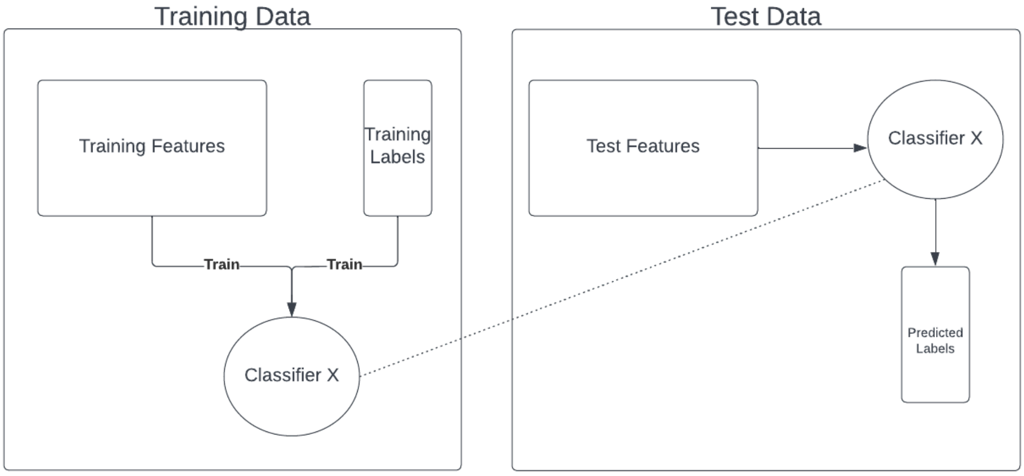


Figure 6.1: Standard Procedure

6.1.2 Unidirectional Context (UC)

After implementing the standard procedure, we implemented a contextual one, where a classifier is used to predict the language of a word based on not only the word itself, but also the predicted language of the previous one.

This is done by first training a classifier with the featurised training data with the addition of the labels of each word's predecessor as a new feature. This means each prediction is made with the information given by a particular feature vector, but also the previous word's prediction.

Then, the same classifier is used on the test features. The first label is assumed to be 0, and the next label is given by the current word's prediction. This is appended to the next row in the features. This is demonstrated in Figure 6.2.

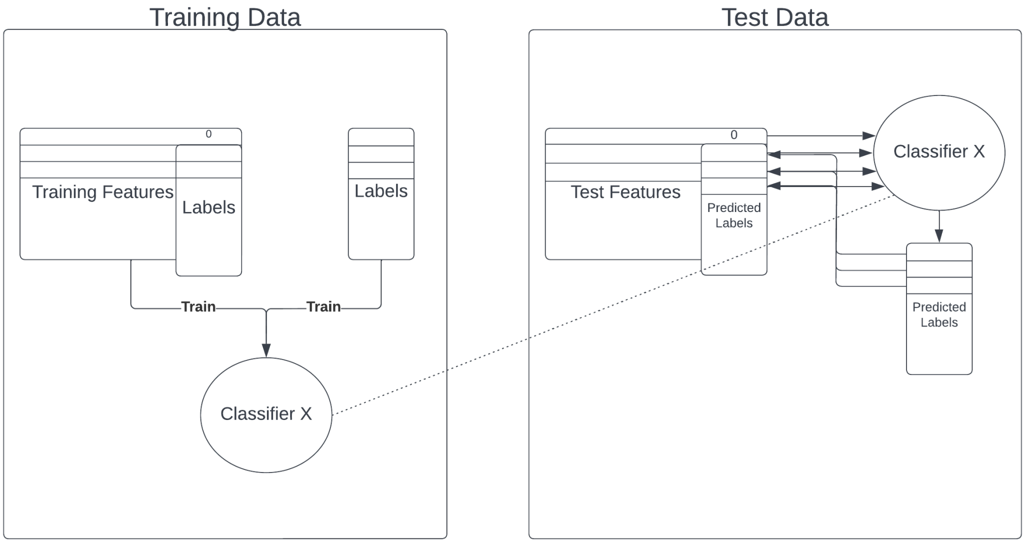


Figure 6.2: UC procedure, a classifier is trained with previous labels as a feature. This classifier then predicts the test data inductively, assuming the first label is 0

6.1.3 Bidirectional Context (BC)

The bidirectional procedure is similar to the unidirectional one, except there are two different classifiers trained with the featurised training data. Classifier A is trained with the standard model, and classifier B is trained with the labels as features twice, one offset forward and one backward. This means each row in the features has the previous label and the next label.

Then the featurised test data is classified using classifier A. The predicted labels resulting from this were appended to the feature vector (in the same fashion as the training data) and classifier B is used to predict the final labels. This process is shown in Figure 6.3.

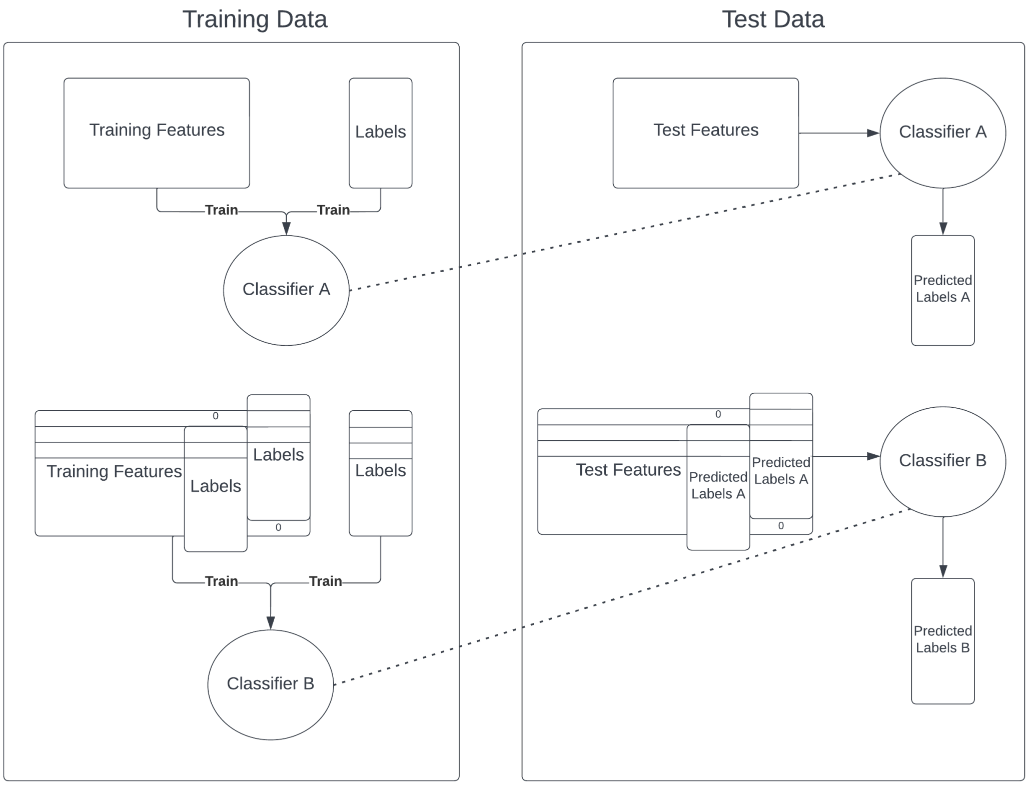


Figure 6.3: BC Procedure, two classifiers are trained. Classifier A is used to predict Labels A, and Classifier B uses them to make a second round of prediction, resuling in Labels B

6.2 Classifiers

With the procedures implemented, the task of classification was performed using the scikit-learn library. A number of different classifiers were imported, based on their reported success in this field. By using existing implementations of machine learning classifiers, we were able to conduct an experiment to find the best performing classifier for this task. The experiment is outlined in Section 6.4.

Gaussian Naive Bayes (GNB)

GNB is a supervised classifier, that uses Bayes theorem (Equation 6.1) to calculate posterior probabilities of classes occurring [24]. It assumes a Gaussian (Equation 6.2) distribution of continuous data, which explains why it is not suited to this task; the data has a significant bias towards German. However, it is included in the experiment for comparison purposes, as it is a simple and commonly used classifier.

$$p(x \mid y) = \frac{p(x)\,p(y \mid x)}{p(y)} \tag{6.1}$$

Where $p(a)$ is the probability of a occurring;
$p(a \mid b)$ is the probability of a occurring, given b;

for any events a, b.

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \tag{6.2}$$

Where σ is the standard deviation; μ is the mean.

Support Vector Machines (SVM)

SVM is a kernel based classifier, that attempts to find the best hyperplane to divide the classes. This is defined by the maximum distance between the plane and the support vectors, which are the points closest to it [8, 10]. It is robust in that small changes in data do not cause big changes in the outcome. SVM is used frequently and successfully in the field of language identification [13], and is included for this reason.

Multilayer Perceptron (MLP)

MLP is a neural network based classifier, and tends to perform well in conventional language identification in general [13]. It assumes initial weights of nodes and then uses backpropogation to adjust them [10]. Gradient descent (Equation 6.5) is utilised to change the weights based on the error $E(n)$, given by Equation 6.3.

$$E(n) = \frac{1}{2} \sum_j e_j^2(n). \tag{6.3}$$

$$e_j(n) = d_j(n) - y_j(n) \tag{6.4}$$

Where $d_j(n)$ is the target value;
$e_j(n)$ is the predicted value.

Stochastic Gradient Descent (SGD)

SGD is an iterative optimisation algorithm, and is a component in other classifiers such as SVM and MLP [25]. SGD attempts to minimise the objective function, for instance the error given by Equation 6.3. This is done iteratively, by first choosing an initial vector w and then stochastically (randomly) choosing samples. Equation 6.5 is used to find an approximate minimum, and this is repeated until the w converge.

$$w = w - \frac{\eta}{n} \sum_{i=1}^{n} \nabla Q_i(w) \tag{6.5}$$

Where η is the learning rate;
n is the number of samples;
$Q_i(w)$ is the objective function given parameter w.

Decision Tree

Decision Tree classification is a simple method that constructs a decision tree, an example of which is given in (Figure 6.4). This method tends to be less robust than more sophisticated classifiers such as SVM. However, decision trees are very effective at feature selection [29].

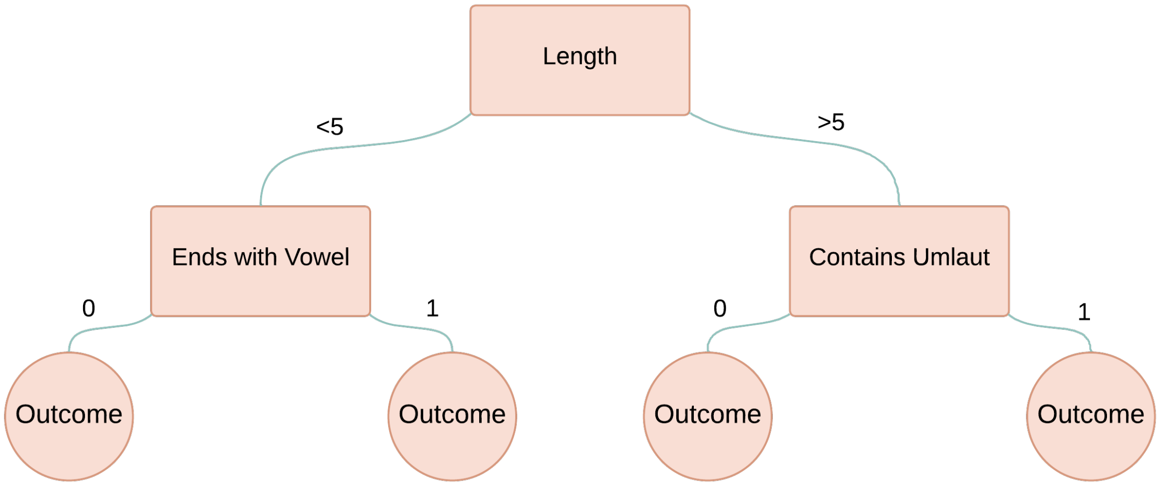


Figure 6.4: Decision Tree, nodes are features and leaves are outcomes. Branches are specific feature values or ranges (e.g 0 or 1)

Random Forest

Random Forest is a method which utilises multiple decision trees and find the most chosen class. An element of randomness is introduced to cause variance in the trees [6]. Using multiple trees allows this method to achieve higher accuracies and robustness than a single tree, at the cost of interpretability and performance. It was chosen due to its reported success in the task of word-level language identification [34].

6.3 Example Use

Using the RF classifier with the standard procedure, we tested the following example data:

Input

"Ich hab den ganzen tag geschlafen, what a pain" ("I slept the whole day, what a pain")

"Ich liebe the new stories, aber nichts beats the old" ("I love the new stories, but nothing beats the old")

Output

In this example data, it incorrectly classifies one word, "what" as German, but correctly identifies the rest. Therefore this classifier identifies the example data with accuracy of 94%.

Word	Ich	hab	den	ganzen	tag	geschläft,	what	a	pain
Predicted Language	0	0	0	0	0	0	0	1	1

Word	Ich	liebe	the	new	stories,	aber	nichts	beats	the	old
Predicted Language	0	0	0	1	1	0	0	1	1	1

6.4 Experiment

The input data for this experiment was in the form of a 4 element tuple:

$$(\texttt{X\_train}, \texttt{X\_test}, \texttt{Y\_train}, \texttt{Y\_test})$$

To determine the best performing classifiers and procedures, we conducted a small experiment. Each imported classifier was tested with each procedure and evaluated.

Before any of this data was passed into the classification procedures, `X_train` and `X_test` were scaled, meaning each of the feature values were scaled between 0 and 1. Next, the data was separated: `Y_test` was used exclusively for evaluation and the other three variables were used for training and classification. Since there were thee procedures and six classifiers, this resulted in a total of 18 combinations that were tested.

The result of each combination was a list of predicted labels. In each instance, these were compared to the true labels and the performance metrics were computed. Since there were more than two classes, the precision, recall and F1-score were all calculated as weighted averages across each class using Equation 6.6.

$$Weighted\ Score = \frac{1}{N}\sum_{i=0}^{m-1} samples_i \cdot score_i \tag{6.6}$$

Where m is the number of classes;
N is the total number of samples;
$samples_i$ is the number of samples for class i;
$score_i$ is the score for class i (e.g F1-score).

The weighted F1-scores and accuracies were collected together can can be seen in Tables 7.3, and 7.4. Additionally, for the best performing combination, further metrics are shown and compared to the baselines of the dictionary and naive n-gram approach. This can be seen in Table 7.2.

Chapter 7

Results and Discussion

The highest F1-Score (90.79%) was achieved using the Random Forest classifier and the UC procedure (RF+UC). Table 7.1 shows the confusion matrix for this result, and Table 7.2 presents further performance metrics, and a comparison to two different baselines.

Table 7.1: Confusion Matrix for RF+UC

		Predicted		
		German	English	Other
True	German	805	51	21
	English	11	52	5
	Other	8	4	28

Table 7.2: RF+UC compared to baselines

Method	Precision	Recall	Accuracy	F1-score
RF+UC	92.44	89.85	89.85	90.79
Frequency	/	/	68.43	67.13
Dict	/	/	50.96	40.44

The best performing classifier for this task was Random Forest. It consistently outperformed the other classifiers in both accuracy and F1-score (Tables 7.3 and 7.4). However, both the Random Forest and Decision Tree classifiers did not benefit from the UC and BC procedures. This is perhaps due to their highly selective nature, which puts more emphasis on the important features. Despite not benefiting from the additional classification procedures, it outperformed the rest.

SVM, performed very well for this task at around 87% accuracy and 90% F1-score. Being a robust classifier, it is logical that its performance is consistent between each procedure. Though the accuracy scores are significantly lower than those of Random Forest, the F1-scores are almost on par. Contrastingly, MLP's performance improved significantly (∼3% difference in accuracy/F1-score) using the UC and BC procedures. This means its performance is comparable to SVM, as long as a contextual procedure is utilised. The SGD classifier, despite being a more primitive method than SVM, performs equally well in F1-score and only loses accuracy using the standard procedure.

GNB did not perform well on this task, likely due to the large bias in the dataset, which it could not overcome. The naive classifier using only character n-gram frequencies (7.2) outperforms GNB considerably.

Table 7.3: Accuracy Scores of chosen classifiers with each procedure

Classifier	Accuracy (%)		
	Standard	UC	BC
Random Forest	90.44	90.79	90.49
SVM	89.78	89.84	90.13
MLP	86.98	89.83	89.69
SGD	88.5	89.21	89.08
Decision Tree	82.19	81.8	83.25
GaussianNB	52.2	52.31	52.31
Mean	85.93	86.87	86.88

Ordered by BC score. Mean does not include GNB

Table 7.4: F1 Scores of chosen classifiers with each procedure

Classifier	F1 Score (%)		
	Standard	UC	BC
Random Forest	90.44	90.79	90.49
SVM	89.78	89.84	90.13
MLP	86.98	89.83	89.69
SGD	88.5	89.21	89.08
Decision Tree	82.19	81.8	83.25
GaussianNB	52.2	52.31	52.31
Mean	87.53	88.41	88.44

Ordered by BC score. Mean does not include GNB

Regarding the classification procedures, there were only small differences in performance. On average, the UC procedure outperforms the standard one by ∼1% in accuracy and F1-score. The BC procedure shows a negligible performance improvement compared to UC. This suggests that the inclusion of context is much more significant for the previous word than the next word. Another possibility is that both sides of the context are important, but UC is on par with BC purely because of its linear "chain" of prediction. This means that each decision is influenced by the previous, which goes on to influence the next. However, with BC this does not occur; the predicted labels are decided all in one instance, and are then used within the feature vector for the second round of prediction. To determine which possibility is the case, one would need to construct a third procedure, which uses a hybrid approach. UC could be applied for preceding labels and BC could be applied to the succeeding labels.

Table 7.2 shows that the best result has a higher precision than recall. As shown in section 2.3.2, it can be beneficial to identify whether precision or recall is more important in a given task. In the case of language identification, it is not clear whether mislabelling words is more or less consequential than under-labelling a language. For this reason we consider precision and recall equally important for this task, making the F1-score (which values them equally) a good performance metric. Though accuracy can be misleading (see section 2.3.2), we include it because in general, it does not differ greatly from the F1-score (indicating that it is not misleading in this case). Moreover, accuracy is an easy metric to understand and is found commonly in literature.

Chapter 8

Conclusions

8.1 Evaluation

As shown in the results section, the best accuracy achieved for this task, was 89.85%. To evaluate its success, we must compare this to a suitable baseline. Because this task has not yet been implemented, we compared our best performing approach to two baselines: a basic dictionary classifier and a basic character n-gram frequency classifier (Table 7.2).

The dictionary classifier, which used the same word lists as the full approach achieved an F1-score of 40%. The reason for this was likely the nature of our data, which came from real social media text, and thus many of its words were not present in the common word lists. Our best performing combination improves the dictionary approach by 50% (F1-score). The n-gram frequency classifier achieves an F1-score of 67%, which indicates that it is a highly useful feature. RF+UC improves this score by 23%. Considering that character n-gram frequencies are the basis for many solutions to this task, we consider it to be a suitable baseline to compare our approach to. An improvement of 23% constitutes a considerable contribution to this task.

Furthermore, the best accuracy score is on par with certain works that address the same task, but with different language pairs [38, 34]. Although a direct comparison made between distinct language pairs is not entirely fair, it can give an estimate of expected performance. The fact that our best scores compare to those of published works in the same field gives merit to our contribution. As discussed in Section 2.2, some languages pairs are more similar than others, and we make the case that German and English are especially similar and so their differentiation is particularly challenging. Because our system was able to identify between English and German in real social media text with high accuracy, we believe this project was very successful.

However, there were certainly limitations in this project: Works by Nguyen et al., who identify between Turkish and Dutch [26] and Mave et al. who identify between Hindi and English [21], achieved higher accuracies of 98% and 96% respectively. There were differences in approach that are responsible for the better accuracies. These include more detailed textual and statistical features for modelling text data, as well as further classification methods. Many successful systems in word level language identification make use of CRFs which is a sophisticated model, obeying the Markov property[1]. Though we implement contextual procedures, these are likely

[1] Each node is dependent only on its neighbours

not as effective as CRFs.

Finally, the main limitation in this project was the size of the dataset. Since it was collected directly from social media, it had to be processed and annotated manually. A public corpus or dataset would have allowed us to use far more data, but for this language pair, one with naturalistic code switching did not exist. For reference, Nguyen et al. [26] use around 11000 words for their test set, compared to our 985 words. For this reason we must concede that the evidence towards this project's success is more limited than that of other systems. However, the method and achievements of this work show promise in this field.

8.2 Future Work

There are numerous possibilities for future work in this domain. Firstly, given more time, a larger dataset should be collected and utilised. Not only will this improve performance, it will provide more evidence as to the success of this system. Additionally, further modelling and classification methods can be implemented and tested, to further improve accuracy and achieve better results. Regarding baselines, work has already shown that grouped language identification does not perform well on a word-level [7], but it could be used to compare performance.

Finally, to prove the use case of this work, one can conduct an experiment with the following steps:

1. Test a NLP task such as sentiment analysis on German social media data and record the results.
2. Use word level language identification demonstrated by this project to tag each word with a language.
3. Use a machine translator to translate each word into a single language, where the source language is given by the predicted label from the previous step
4. Test the same NLP task using this newly translated data, and compare performance.

This would demonstrate the main use case of this project as social media text often contains code switching. The system developed in this project is able to tag each word of code switched text with a language, allowing it to be normalised and used more successfully in other NLP tasks.

Bibliography

[1] "1,000 most common US English words." [Online]. Available: https://gist.github.com/deekayen/4148741

[2] "1000-most-common-words/1000-most-common-german-words.txt at master · CodeBrauer/1000-most-common-words." [Online]. Available: https://github.com/CodeBrauer/1000-most-common-words/blob/master/1000-most-common-german-words.txt

[3] "List of countries by English-speaking population - Wikipedia." [Online]. Available: https://en.wikipedia.org/wiki/List_of_countries_by_English-speaking_population

[4] "Scheduled languages in descending order of speakers' strength," 2011. [Online]. Available: https://censusindia.gov.in/2011census/Language-2011/Statement-4.pdf

[5] T. Baumann, A. Köhn, and F. Hennig, "The Spoken Wikipedia Corpus collection: Harvesting, alignment and an application to hyperlistening," *Language Resources and Evaluation*, vol. 53, no. 2, pp. 303–329, 2019. [Online]. Available: https://www.idiap.ch/en/dataset/code-switching

[6] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001. [Online]. Available: https://doi.org/10.1023/A:1010933404324

[7] Ö. Çetinoğlu, S. Schulz, and T. Vu, "Challenges of Computational Processing of Code-Switching," in *Proceedings of the Second Workshop on Computational Approaches to Code Switching*. Austin, TX, USA: ACL, 2016, pp. 1–11. [Online]. Available: https://aclanthology.org/W16-5801/

[8] O. Chapelle, "Training a Support Vector Machine in the Primal," *Neural Computation*, vol. 19, no. 5, pp. 1155–1178, 2007. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/6796736

[9] N. Chomsky, "Linguistics and cognitive science: problems and mysteries," *The Chomskyan Turn*, p. 23, 1991.

[10] R. Collobert and S. Bengio, "Links between Perceptrons, MLPs and SVMs," in *Proceedings of the Twenty-First International Conference on Machine Learning*, ser. ICML '04. New York, USA: ACM, 2004, p. 23. [Online]. Available: https://doi.org/10.1145/1015330.1015415

[11] A. Das and . B. Gambäck, "Code-Mixing in Social Media Text: The Last Language Identification Frontier?" *Revue TAL*, vol. 54, no. 3, pp. 41–64, 2016. [Online]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2391067

[12] C. M. Eastman, "Monica Heller (ed.), Codeswitching: Anthropological and sociolinguistic perspectives (Contributions to the Sociology of Language 48). Berlin: Mouton de Gruyter, 1988. Pp. 278." in *Language in Society*. Cambridge University Press, 1990, vol. 19, no. 3, pp. 442–447.

[13] A. Garg, V. Gupta, and M. Jindal, "A Survey of Language Identification Techniques and Applications," *Journal of Emerging Technologies in Web Intelligence*, vol. 6, no. 4, 2014. [Online]. Available: https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.689.5252&rep=rep1&type=pdf

[14] G. S. Handelman, H. K. Kok, R. V. Chandra, A. H. Razavi, S. Huang, M. Brooks, M. J. Lee, and H. Asadi, "Peering Into the Black Box of Artificial Intelligence: Evaluation Metrics of Machine Learning Methods," *American Journal of Roentgenology*, vol. 212, no. 1, pp. 38–43, oct 2018.

[15] R. R. Heredia and J. Altarriba, "Bilingual Language Mixing: Why Do Bilinguals Code-Switch?" *Current Directions in Psychological Science*, vol. 10, no. 5, pp. 164–168, 2001. [Online]. Available: https://doi.org/10.1111/1467-8721.00140

[16] N. Kratzke, "Monthly Samples of German Tweets," feb 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3633935

[17] S. Krüger, "Warum Denglisch Sprachmüll ist," nov 2007. [Online]. Available: https://www.welt.de/vermischtes/article1366422/Warum-Denglisch-Sprachmuell-ist.

[18] J. MacSwan, "Code-Switching and Grammatical Theory," in *The Handbook of Bilingualism and Multilingualism, 2nd Edition*, 2nd ed., T. K. Bhatia and W. C. Ritchie, Eds. Wiley-Blackwell, 2012, pp. 283–308. [Online]. Available: https://learning.oreilly.com/library/view/the-handbook-of/9781118332412/c13.xhtml#c13-sec1-0002

[19] S. Maharjan, E. Blair, S. Bethard, and T. Solorio, "Developing Language-tagged Corpora for Code-switching Tweets," in *Proceedings of The 9th Linguistic Annotation Workshop*. Denver, Colorado, USA: ACL, jun 2015, pp. 72–84. [Online]. Available: https://aclanthology.org/W15-1608

[20] P. Mathur, R. Shah, R. Sawhney, and D. Mahata, "Detecting Offensive Tweets in Hindi-English Code-Switched Language," in *Proceedings of the Sixth International Workshop on Natural Language Processing for Social Media*. Melbourne, Australia: ACL, jul 2018, pp. 18–26. [Online]. Available: https://aclanthology.org/W18-3504

[21] D. Mave, S. Maharjan, and T. Solorio, "Language Identification and Analysis of Code-Switched Social Media Text," in *Proceedings of The Third Workshop on Computational Approaches to Code-Switching*. Melbourne, Australia: ACL, 2018, pp. 51–61. [Online]. Available: https://aclanthology.org/W18-3206.pdf

[22] M. L. McHugh, "Interrater reliability: the kappa statistic," *Biochemia medica*, vol. 22, no. 3, pp. 276–282, 2012. [Online]. Available: https://pubmed.ncbi.nlm.nih.gov/23092060

[23] P. McNamee, "Language Identification: A Solved Problem Suitable for Undergraduate Instruction," *J. Comput. Sci. Coll.*, vol. 20, no. 3, pp. 94–101, feb 2005. [Online]. Available: https://dl.acm.org/doi/10.5555/1040196.1040208

[24] K. P. Murphy and Others, "Naive bayes classifiers," *University of British Columbia*, vol. 18, no. 60, pp. 1–8, 2006. [Online]. Available: https://www.ic.unicamp.br/$\sim$rocha/teaching/2011s1/mc906/aulas/naive-bayes.pdf

[25] P. Netrapalli, "Stochastic Gradient Descent and Its Variants in Machine Learning," *Journal of the Indian Institute of Science*, vol. 99, no. 2, pp. 201–213, jun 2019. [Online]. Available: https://link.springer.com/article/10.1007/s41745-019-0098-4

[26] D.-P. Nguyen and A. S. Dogruoz, "Word level language identification in online multilingual communication," in *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*. Seattle, WA, USA: ACL, oct 2013, pp. 857–862. [Online]. Available: https://aclanthology.org/D13-1084.pdf

[27] K. Pant and T. Dadu, "Towards Code-switched Classification Exploiting Constituent Language Resources," in *Proceedings of the 1st Conference of the Asia-Pacific Chapter of the Association for Computational Linguistics and the 10th International Joint Conference on Natural Language Processing: Student Research Workshop*. Suzhou, China: ACL, dec 2020, pp. 37–43. [Online]. Available: https://aclanthology.org/2020.aacl-srw.6

[28] P. Patwa, G. Aguilar, S. Kar, S. Pandey, S. PYKL, B. Gambäck, T. Chakraborty, T. Solorio, and A. Das, "SemEval-2020 Task 9: Overview of Sentiment Analysis of Code-Mixed Tweets," in *Proceedings of the Fourteenth Workshop on Semantic Evaluation*. Barcelona: ICCL, dec 2020, pp. 774–790. [Online]. Available: https://aclanthology.org/2020.semeval-1.100

[29] J. R. Quinlan, "Induction of decision trees," *Machine Learning 1986 1:1*, vol. 1, no. 1, pp. 81–106, mar 1986. [Online]. Available: https://link.springer.com/article/10.1007/BF00116251

[30] Y. Samih, S. Maharjan, M. Attia, L. Kallmeyer, and T. Solorio, "Multilingual Code-switching Identification via LSTM Recurrent Neural Networks," in *Proceedings of the Second Workshop on*

Computational Approaches to Code Switching. Austin, Texas: ACL, nov 2016, pp. 50–59. [Online]. Available: https://aclanthology.org/W16-5806

[31] M. Sanad, Z. Rizvi, A. Srinivasan, T. Ganu, M. Choudhury, and S. Sitaram, "GCM: A Toolkit for Generating Synthetic Code-mixed Text," in *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics*. ACL, 2021, pp. 205–211. [Online]. Available: https://aclanthology.org/2021.eacl-demos.24.pdf

[32] Y. Sasaki and R. Fellow, "The truth of the F-measure," 2007. [Online]. Available: https://www.cs.odu.edu/$\sim$mukka/cs795sum09dm/Lecturenotes/Day3/F-measure-YS-26Oct07.pdf

[33] H. Schendl, "Code-switching in early English literature," *Language and Literature*, vol. 24, no. 3, pp. 233–248, 2015. [Online]. Available: https://journals.sagepub.com/doi/pdf/10.1177/0963947015585245

[34] K. Shanmugalingam and S. Sumathipala, "Language identification at word level in Sinhala-English code-mixed social media text," in *Proceedings - IEEE International Research Conference on Smart Computing and Systems Engineering, SCSE 2019*. Colombo, Sri Lanka: IEEE, mar 2019, pp. 113–118. [Online]. Available: https://ieeexplore.ieee.org/document/8842795

[35] R. M. K. Sinha and A. Thakur, "Machine Translation of Bi-lingual Hindi-English (Hinglish) Text," in *Proceedings of Machine Translation Summit X: Papers*, Phuket, Thailand, sep 2005, pp. 149–156. [Online]. Available: https://aclanthology.org/2005.mtsummit-papers.20

[36] H. Takçi and E. Ekinci, "Minimal feature set in language identification and finding suitable classification method with it," *Procedia Technology*, vol. 1, pp. 444–448, 2012. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2212017312001004

[37] I. Tarunesh, S. Kumar, and P. Jyothi, "From Machine Translation to Code-Switching: Generating High-Quality Code-Switched Text," in *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing*. ACM, 2021, pp. 3154–3169. [Online]. Available: https://aclanthology.org/2021.acl-long.245.pdf

[38] C. Voss, S. Tratz, J. Laoudi, and D. Briesch, "Finding Romanized Arabic Dialect in Code-Mixed Tweets," in *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC'14)*. Reykjavik, Iceland: ELRA, 2014, pp. 2249–2253. [Online]. Available: http://www.lrec-conf.org/proceedings/lrec2014/pdf/1116_Paper.pdf

[39] G. I. Winata, A. Madotto, C.-S. Wu, and P. Fung, "Code-Switched Language Models Using Neural Based Synthetic Data from Parallel Sentences," in *Proceedings of the 23rd Conference on Computational Natural Language Learning (CoNLL)*. Hong Kong, China: ACL, nov 2019, pp. 271–280. [Online]. Available: https://aclanthology.org/K19-1026

[40] J. Xu and F. Yvon, "Can You Traducir This? Machine Translation for Code-Switched Input," in *Proceedings of the Fifth Workshop on Computational Approaches to Linguistic Code-Switching*. Online: ACL, jun 2021, pp. 84–94. [Online]. Available: https://aclanthology.org/2021.calcs-1.11

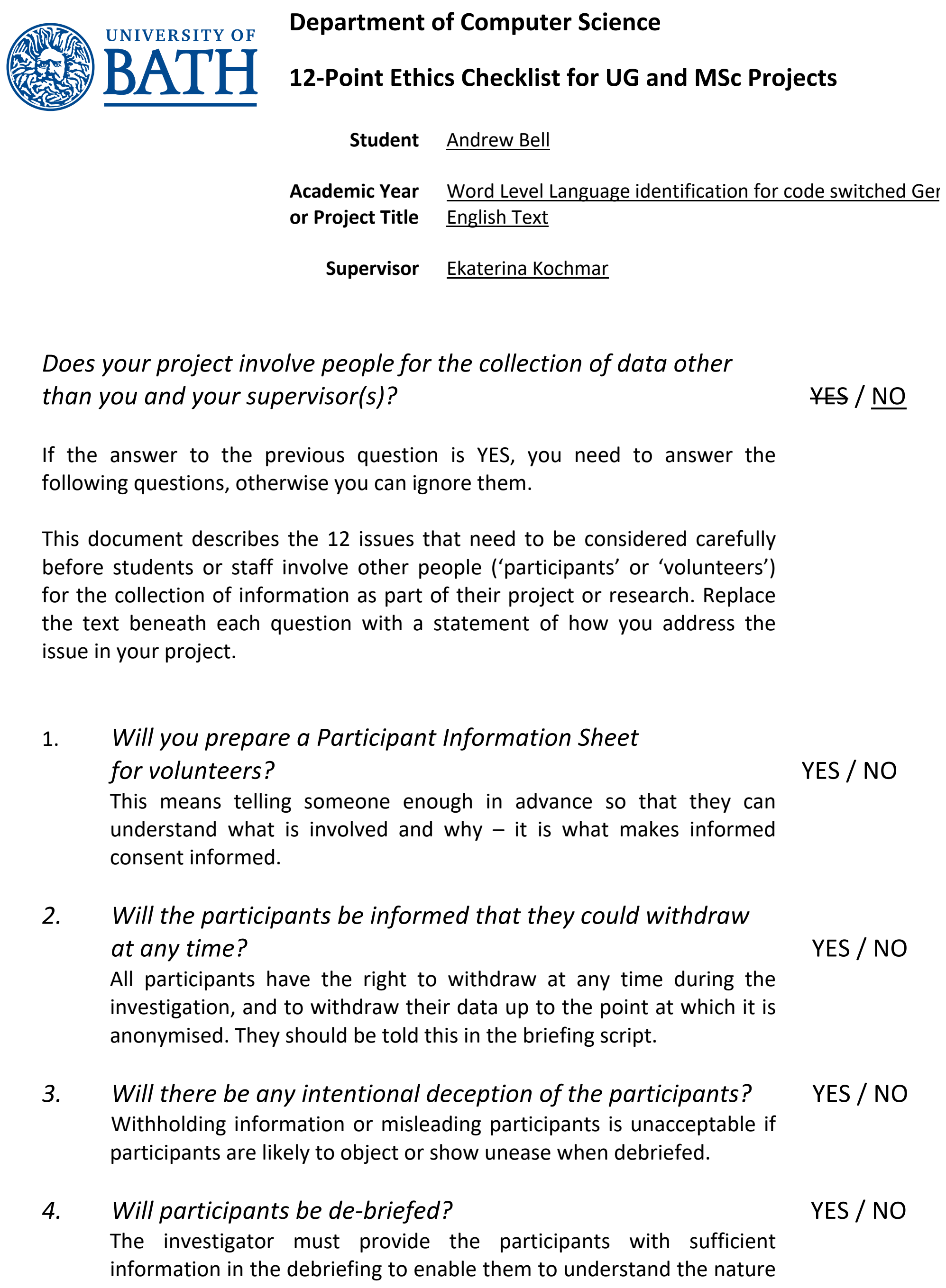
[41] X. Yang and W. Liang, "An N-Gram-and-Wikipedia joint approach to Natural Language Identification," *2010 4th International Universal Communication Symposium*, pp. 332–339, 2010. [Online]. Available: https://ieeexplore.ieee.org/document/5666010

Appendix A

Ethics

All user data is stored according to the Twitter's privacy policy. No information other than Twitter activity is associated with stored user identification numbers. All API keys are stored securely and are not shared in this work. Tweets are stored without any links, hashtags or user mentions.

https://developer.twitter.com/en/developer-terms/agreement-and-policy



Department of Computer Science

12-Point Ethics Checklist for UG and MSc Projects

Student Andrew Bell

Academic Year or Project Title Word Level Language identification for code switched Ger
English Text

Supervisor Ekaterina Kochmar

Does your project involve people for the collection of data other than you and your supervisor(s)? ~~YES~~ / NO

If the answer to the previous question is YES, you need to answer the following questions, otherwise you can ignore them.

This document describes the 12 issues that need to be considered carefully before students or staff involve other people ('participants' or 'volunteers') for the collection of information as part of their project or research. Replace the text beneath each question with a statement of how you address the issue in your project.

1. *Will you prepare a Participant Information Sheet for volunteers?* YES / NO
   This means telling someone enough in advance so that they can understand what is involved and why – it is what makes informed consent informed.

2. *Will the participants be informed that they could withdraw at any time?* YES / NO
   All participants have the right to withdraw at any time during the investigation, and to withdraw their data up to the point at which it is anonymised. They should be told this in the briefing script.

3. *Will there be any intentional deception of the participants?* YES / NO
   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

4. *Will participants be de-briefed?* YES / NO
   The investigator must provide the participants with sufficient information in the debriefing to enable them to understand the nature

Appendix B

Annotation Instructions

This is 10% of the dataset, please annotate each word according to the following instructions:

0 = German

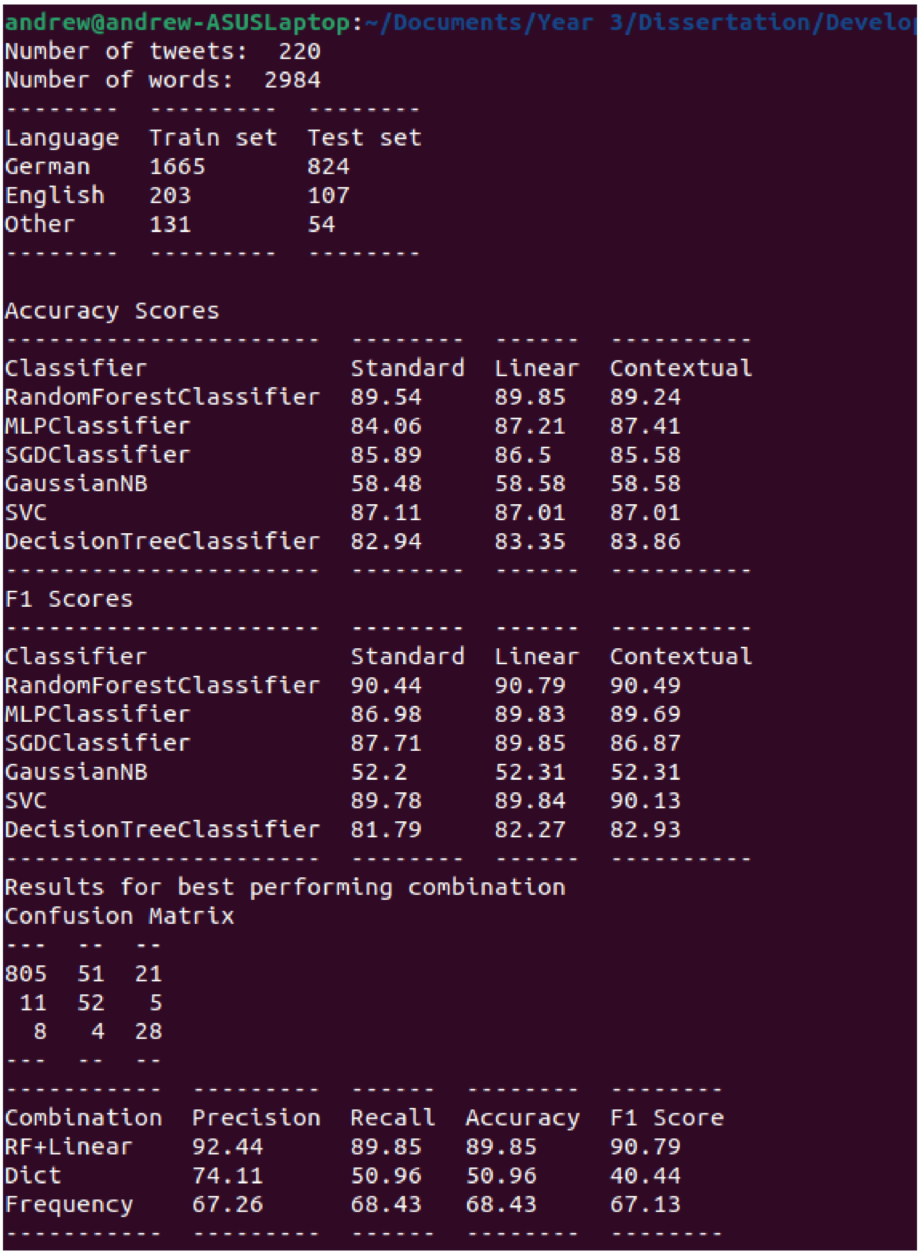
1 = English

2 = Other

Any time you are unsure of whether to call a word German or English, please decide based on the context. If a word is best classified as neither English nor German, please select Other (2). Write the number in the square brackets next to each word.

Appendix C

Raw Results Output



```
andrew@andrew-ASUSLaptop:~/Documents/Year 3/Dissertation/Develop
Number of tweets:  220
Number of words:  2984
--------  ---------  --------
Language  Train set  Test set
German    1665       824
English   203        107
Other     131        54
--------  ---------  --------

Accuracy Scores
----------------------  --------  ------  ----------
Classifier              Standard  Linear  Contextual
RandomForestClassifier  89.54     89.85   89.24
MLPClassifier           84.06     87.21   87.41
SGDClassifier           85.89     86.5    85.58
GaussianNB              58.48     58.58   58.58
SVC                     87.11     87.01   87.01
DecisionTreeClassifier  82.94     83.35   83.86
----------------------  --------  ------  ----------
F1 Scores
----------------------  --------  ------  ----------
Classifier              Standard  Linear  Contextual
RandomForestClassifier  90.44     90.79   90.49
MLPClassifier           86.98     89.83   89.69
SGDClassifier           87.71     89.85   86.87
GaussianNB              52.2      52.31   52.31
SVC                     89.78     89.84   90.13
DecisionTreeClassifier  81.79     82.27   82.93
----------------------  --------  ------  ----------
Results for best performing combination
Confusion Matrix
---  --  --
805  51  21
 11  52   5
  8   4  28
---  --  --
-----------  ---------  ------  --------  --------
Combination  Precision  Recall  Accuracy  F1 Score
RF+Linear    92.44      89.85   89.85     90.79
Dict         74.11      50.96   50.96     40.44
Frequency    67.26      68.43   68.43     67.13
-----------  ---------  ------  --------  --------
```

Appendix D

Code and Data

In this appendix, the most important files are included. “findusers.py” and “tweetfetch.py” are part of the data retrieval process, and “model.py” and “classify.py” contain the code for the modelling implementations and the classification experiments respectively.

All the code and data can be found in the compressed folder uploaded with this dissertation. The raw data can be found in the following two files:

- Development/Data/Collection/usertweets.txt
- Development/Data/Collection/usertweets2.txt

D.1 File: findusers.py

```
#This file filters the tweets in a given file, and extracts the
    users who wrote those filtered tweets

#Input = .json file given by system argument: List of tweets and
    users
#Output = usrs.txt: List of users who wrote "code switched" tweets


import json
import sys

filename = sys.argv[1]
infile=open(filename,'r')
dictfile=open('words.txt','r')


data = json.load(infile)
en_words = dictfile.readlines()


def contains_word(s, w):
    return (' ' + w + ' ') in (' ' + s + ' ')

users = set()
for field in data:
    try:
        text = field['text']
        user= field['user']
        include=False
        with open('usrs.txt','a') as outfile:
            for w in en_words:
                if contains_word(text.lower(),w.lower()):
                    include=True
                    print("word matched: ", w)
                    break;
            if include:
                if user not in users:
                    print("adding user", user)
                    users.add(user)
                    outfile.write(user)
                    outfile.write("\n")
                else:
                    print("duplicate user: ", user)
    except KeyError:
        continue
infile.close()
dictfile.close()
outfile.close()
```

D.2 File: tweetfetch.py

```
#Gets tweets from users
#Input = usrs.txt : list of users
#Output = usertweets.txt: list of tweets published by the
    specified users

import os
import tweepy as tw
import pandas as ps
import re
userid = "904787255868391424"
rejectwords = ['www.', 'http', '@', '#']


def twitter_auth():
    #Information here is redacted for submission
    consumer_key = "X"
    consumer_secret = "X"
    access_token = "X"
    access_secret = "X
    auth = tw.OAuthHandler(consumer_key,consumer_secret)
    auth.set_access_token(access_token,access_secret)
    return auth

def get_twitter_client():
    auth = twitter_auth()
    client = tw.API(auth)
    return client

out = open("usertweets.txt", 'a')
userfile = open("usrs.txt", 'r')
users = userfile.readlines()
client = get_twitter_client()

for u in users:
    tweets= []
    try:
        tweets =
            client.user_timeline(user_id=u,count=200,include_rts=False,tweet_mode=
            'extended')
    except:
        print("error for ", u)
    for info in tweets:
        if info.lang == 'de':
            text = info.full_text
            i=0
            listwords = text.split()
            for word in text.split():
                for r in rejectwords:
                    if word.startswith(r):
                        listwords[i]=''
                i+=1
            string1= " "
            string1 = string1.join(listwords)
            out.write(string1)
            out.write("\n")


out.close()
userfile.close()
```

D.3 File: model.py

```
#Contains various functions to model text on a word level

import pandas as pd
import numpy as np

vowels = ['a','e','i','o','u']
umlauts = ['ä','ü','ö','ß']
german = 0
english = 1
other = 2

#Extracts a list of character n-grams from a given word
def character_ngrams(word,n):
    word = word.strip().lower()
    result = []
    i=0
    for i in range (0,len(word)):
        ngram = ""
        append = True
        for j in range (0,n):
            if (i+j>=len(word) or not word[i+j].isalpha()):
                append = False
                break;
            ngram+=word[i+j]
        if (append):
            result.append(ngram)
    return result

#Creates a hash table, storing the frequency of the character
    ngrams of the given data
def ngram_table(dict, total, data, n, weight):
    for word in data:
        ngrams = character_ngrams(word, n)
        i = n-1
        while (len(ngrams)==0 and i>0): # If no ngrams are
            generated, use (n-1)grams until some are found
            ngrams=character_ngrams(word,i)
            i = i-1
        for ng in ngrams:
            total+=1
            if ng in dict:
                dict[ng]+=weight
            else:
                dict[ng]=weight
    return dict, total


#Given a word and frequency table, extracts ngrams and returns the
    sum of their frequency scores
def frequency_score(word, freqDict, n):
    ngrams = character_ngrams(word,n)
    i= n-1
    while (len(ngrams)==0 and i>0): # If no n-grams are generated,
        use (n-1)grams until some are found
        ngrams=character_ngrams(word,i)
        i = i-1
    score=0
    for ng in ngrams:
        if (ng in freqDict):
            score+=freqDict[ng]
    return score

def frequency_predict(word,model, n):
    german_score = frequency_score(word,model[german],2)
    english_score = frequency_score(word,model[english],2)
    return (english_score>german_score)
```

```
def contains_word(s, w):
    return (' ' + w + ' ') in (' ' + s + ' ')

def begins_vowel(word):
    return (word[0] in vowels)

def ends_vowel(word):
    return(word[len(word)-1] in vowels)

def contains_apostrophe(word):
    return ("'" in word)

def contains_umlaut(word):
    for c in word:
        if c in umlauts:
            return True
    return False

#Performs simplistic dictionary and search and returns boolean
    values for presense of given word in
# list of german and english words respectively
def dict_scores(word,wordlists):
    word = word.lower()
    de = 0
    en = 0
    for w in wordlists[german]:
        if contains_word(w.strip().lower(), word):
            de = 1
    for w in wordlists[english]:
        if contains_word(w.strip().lower(), word):
            en = 1
    return de,en

#Naively predicts language of word based on dictionary scores
def dict_predict(word,wordlists):
    de, en = dict_scores(word,wordlists)
    if (de):
        return 0
    if (en):
        return 1
    return 2

#Given word and information, return feaure vector for this word
def get_features(word, wordlists, model, position, n):
    length = len(word)
    german_score = frequency_score(word,model[german],n)
    english_score = frequency_score(word,model[english],n)
    is_upper = word.isupper()
    capitalised = word[0].isupper()
    german_dict, english_dict = dict_scores(word,wordlists)
    features = [word, length, german_score, english_score,
        german_dict, english_dict, capitalised, is_upper, position,
                begins_vowel(word), ends_vowel(word),
                    contains_apostrophe(word),
                    contains_umlaut(word)]
    return features

def create_model(X, Y, wordlists):
    n = 3 # N for choosing character n-grams
    de_freq, de_total = ngram_table({},0,wordlists[0],n, 2)
    en_freq, en_total = ngram_table({},0,wordlists[1],n, 2)

    de_X = []
    en_X = []
    i=0
    for word in X:
        if (Y[i]==0):
            de_X.append(word[0])
        elif (Y[i]==1):
            en_X.append(word[0])
        i+=1

    de_freq, de_total = ngram_table(de_freq, de_total, de_X, n, 3)
    en_freq, en_total = ngram_table(en_freq, en_total, en_X, n, 3)

    for i in de_freq:
        de_freq[i] = de_freq[i]/de_total #Divides the frequency
            data by the total number of ngrams to normalise it
                                         #This means two tables
                                             with a different total
                                             number can still be
                                             compared
    for i in en_freq:
        en_freq[i] = en_freq[i]/en_total
    return (de_freq, en_freq)

def model_data(X, wordlists, model):
    n = 3 # N for choosing character n-grams
    features = []
    i=0
    for word in X:
        features.append(get_features(word[0],wordlists, model,
            word[1], n))
        i+=1
    return features
```

D.4 File: classify.py

```
#Uses annotated data, alongside model.py to model and classify
    data, and evaluate perfromance
#Input = annotated2_pickle.txt: Annotated tweets.
    de_words_1000.txt, en_words_1000.txt : word lists for use in
    dictionary and n-grams
#Output = console: information about the data and results tables

from warnings import simplefilter
from sklearn.exceptions import ConvergenceWarning
from sklearn.exceptions import UndefinedMetricWarning
simplefilter("ignore", category=ConvergenceWarning)
simplefilter("ignore", category=UndefinedMetricWarning)

#Import modelling functions from model.py
from model import model_data
from model import create_model
from model import dict_predict
from model import frequency_predict

from copy import copy
from statistics import mean
from sklearn.metrics import accuracy_score, recall_score,
    f1_score, precision_score,confusion_matrix,
    ConfusionMatrixDisplay
from sklearn.preprocessing import MinMaxScaler

#Importing various classifiers from sklearn library
from tabulate import tabulate
import pickle #Pickle used to retrieve data used for training and
    testing
import numpy as np
from copy import copy
from statistics import mean
from sklearn.metrics import accuracy_score, f1_score
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import MinMaxScaler

#Importing various classifiers from sklearn library
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import SGDClassifier

#Preprocessing functions
def removeInitial(a):
    return a[1:]

def flatten(iterable):
    count = 0
    for i in iterable:
        for j in i:
            count+=1
    new = [0 for i in range(count)]
    x = 0
    for i in iterable:
        for j in i:
            new[x]=j
            x+=1
    return new

def flatten_position(iterable):
    count = 0
    for i in iterable:
        for j in i:
            count+=1
    new = [0 for i in range(count)]
    x = 0
    for i in iterable:
        pos = 0
        for j in i:
            new[x]=[j,pos]
            x+=1
            pos+=1
    return new



def print_counts(ytrain,ytest):
    count_en = 0
    count_de = 0
    count_am = 0
    for l in Y_test: #Counts for each label (for the test set),
        produces table for output
        if l==0:
            count_de+=1
        elif l==1:
            count_en+=1
        else:
            count_am+=1

    count_en_train = 0
    count_de_train = 0
    count_am_train = 0
    for l in Y_train: #Counts for each label (for the training
        set), produces table for output
        if l==0:
            count_de_train+=1
        elif l==1:
            count_en_train+=1
        else:
            count_am_train+=1

    table = [["Language", "Train set", "Test set"],
             ["German", count_de_train,count_de],
             ["English",count_en_train,count_en],
             ["Other", count_am_train,count_am ]]

    print (tabulate(table)#,tablefmt='latex_raw')) #Prints word
        counts in nice format

#Naive prediction functions

def predict_dictionary(xtest, wordlists):# Dictionary only
    ypred = []
    for i in X_test:
        ypred.append(dict_predict(i[0],wordlists)) # Uses naive
            dictionary classifier
    return ypred

def predict_frequency(xtest, model, n): #Character n-gram
    frequency only
    ypred = []
    for i in X_test:
        ypred.append(frequency_predict(i[0], model, n)) # Uses
            naive frequency score classifier
    return ypred

#Machine learning prediction functions, take classifier as
    parameter

def predict_standard(xtrain, xtest, ytrain, clf): #Standard model
    clf1 = copy(clf)
    clf1.fit(xtrain,ytrain)
    ypred = clf1.predict(xtest) #Standard prediction using given
        classifer
    return ypred

def predict_linear(xtrain, xtest, ytrain, clf): #Linear model
    xtrain2 = []
    i=0
    for x in xtrain:
        #Append labels to features with a -1 offset
        #i.e each feature contains the label of the previous word
        if (i>0):
            inner = x+[ytrain[i-1]]
        else:
            inner = x+[0]
        xtrain2.append(inner)
        i+=1
    clf2 = copy(clf)
    clf2.fit(xtrain2, ytrain)
    zero = [0]
    ypred = []
    xtest2 = [np.concatenate([x,zero]) for x in xtest] #
        Populating new feature vector with additional column of 0s
    xtest2 = np.array(xtest2)
    xtest2[0][len(xtest2[0])-1]= 0 #First entry is assumed as 0
    i=0
    for x in xtest2:
        pred = clf2.predict([x])
        if i<len(xtest2)-1:
            xtest2[i+1][len(xtest2[0])-1] = pred #Prediction of
                current word is appended to the next feature vector
        ypred.append(pred)
        i+=1

    #Evaluate perfromance
    return ypred

def predict_context(xtrain,xtest,ytrain, clf): #Contextual model
    clf1 = copy(clf)
    clf1.fit(xtrain,ytrain)
    xtrain2 = []
    i=0
    for x in xtrain: # This loop appends the labels to the
        features, with a +1 and -1 offset
                     # i.e each feature now contains the label of
                         the previous and next word
        if (i>1 and i<len(xtrain)-1):
            inner = x+[ytrain[i-1],ytrain[i+1]]
        elif (i>1):
            inner = x+[ytrain[i-1], 0]
        elif (i<len(xtrain)-1):
            inner = x+ [0, ytrain[i+1]]
        else:
            inner = x+[0,0]
        xtrain2.append(inner)
        i+=1

    clf2 = copy(clf)
    clf2.fit(xtrain2, ytrain)

    ytestpred = clf1.predict(xtest) #Get predicted labels using
        the first (standard) classifer
    xtest2 = []
    y_old = 0
    i=0
    for x in xtest:
        #Use the predicted labels to add to the features (in the
            same way as training set)
        if (i>1 and i<len(xtest)-1):
            inner = x+[ytestpred[i-1],ytestpred[i+1]]
        elif (i>1):
            inner = x+[ytestpred[i-1], 0]
        elif (i<len(xtest)-1):
            inner = x+ [0,ytestpred[i+1]]
        else:
            inner = x+[0,0]
        xtest2.append(inner)
        i+=1
    ytestpred2 = clf2.predict(xtest2) #Get final predicted labels
        using the second (contextual) classifier

    return ytestpred2

#Retrieving word lists from files
infile = open('de_words_1000.txt','r')
DE_WORDS = infile.readlines()
infile.close()
infile = open('en_words_1000.txt','r')
EN_WORDS = infile.readlines()
infile.close()
wordlists = (DE_WORDS,EN_WORDS)

#Retrieving annotated data from files
infile = open('annotated2_pickle.txt','rb')
DATA = pickle.load(infile)
infile.close()

print("Number of tweets: ",len(DATA))


labels = DATA['Languages'] # Retrieve language labels from data
sentences = DATA['Words']

# Retrieve tweets from data, and model using model.py. This
    returns the features and the frequency tables (for use in
    frequency classifer)

Y = flatten(labels) # Flatten shape of labels to work in
    classifiers
X = flatten_position(sentences) # Flatten shape of word data but
    retain positions


print("Number of words: ",len(Y))

#Split data into training and test sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
    test_size=0.33, random_state=321)
print_counts(Y_train,Y_test) # Print nice table of all the word
    counts

#Using training data (X and Y) along with word lists to create
    frequency tables, which are the basis of the model
model = create_model(X_train, Y_train, wordlists)

#Storing model for use later
with open('model.txt','wb') as outfile:
     pickle.dump(model,outfile)


X_train_features = model_data(X_train, wordlists, model)
X_train_f = list(map(removeInitial,X_train_features)) #Remove word
    itelf from feature vector (cannot be parsed by classifiers)
scaler = MinMaxScaler().fit(X_train_f) #Scale features
X_train_scaled = scaler.transform(X_train_f)


X_test_features = model_data(X_test, wordlists, model)
X_test_f = list(map(removeInitial,X_test_features)) #Remove word
    itelf from feature vector (cannot be parsed by classifiers)
scaler = MinMaxScaler().fit(X_test_f) #Scale features
```

```
X_test_scaled = scaler.transform(X_test_f)

#Parameter vector to pass to all the prediction functions
params= (X_train_scaled.tolist(), X_test_scaled.tolist(), Y_train)

#Classifiers to test
rf = RandomForestClassifier(max_depth=8, random_state=1)
mlp = MLPClassifier(solver='lbfgs',
    alpha=1e-5,hidden_layer_sizes=(16, 8), random_state=1,
    max_iter=20)
sgd = SGDClassifier(loss="hinge", penalty="l2", max_iter=20)
clfs = [rf, mlp, sgd, GaussianNB(), SVC(),
    DecisionTreeClassifier()]

accuracy_results = [["Classifier", "Standard", "Linear",
    "Contextual"]]
fscore_results= [["Classifier", "Standard", "Linear",
    "Contextual"]]
for clf in clfs: # Test each classifier with each model
    standard = predict_standard(*params,clf)
    sa = accuracy_score(standard,Y_test)
    sf = f1_score(standard,Y_test, average= 'weighted')
    linear = predict_linear(*params,clf)
    la = accuracy_score(linear,Y_test)
    lf = f1_score(linear,Y_test, average= 'weighted')
    context = predict_context(*params,clf)
    ca = accuracy_score(context,Y_test)
    cf = f1_score(context,Y_test, average= 'weighted')

    name = str(type(clf)).split(".")[-1][:-2]
    accuracy_results.append([name,str(round(sa*100,2)),str(round(la*100,2)),str(round(ca*100,2))])
    fscore_results.append([name,str(round(sf*100,2)),str(round(lf*100,2)),str(round(cf*100,2))])


accuracy_results.sort(key= lambda row: row[3], reverse=True)
fscore_results.sort(key=lambda row: row[3], reverse=True)


#Print results
print("\nAccuracy Scores")
print (tabulate(accuracy_results,tablefmt = 'latex_raw'))
print("F1 Scores")
print (tabulate(fscore_results,tablefmt = 'latex_raw'))



#Testing basic dictionary and frequency classifiers
Y_pred_dict = predict_dictionary(X_test, wordlists)
Y_pred_freq = predict_frequency(X_test, model, 3)

#In depth results for best performer
print("Results for best performing combination")

print("Confusion Matrix")
best = predict_linear(*params, copy(rf))
cm = confusion_matrix(best,Y_test, labels = [0, 1, 2])
print(tabulate(cm))#,tablefmt='latex_raw'))

detailed_results =
    [["Combination","Precision","Recall","Accuracy","F1 Score"]]

for vals, name in [(best,"RF+Linear"), (Y_pred_dict,"Dict"),
    (Y_pred_freq,"Frequency")]:
    p = precision_score(vals,Y_test,average='weighted')
    r = recall_score(vals,Y_test,average='weighted')
    a = accuracy_score(vals,Y_test)
    f= f1_score(vals,Y_test, average='weighted')
    detailed_results.append([name,round(p*100,2),round(r*100,2),round(a*100,2),rou

print(tabulate(detailed_results))


example_classifier = copy(rf)
example_classifier.fit(X_train_scaled,Y_train)
with (open("classifier.txt",'wb') as outfile):
    pickle.dump(example_classifier,outfile)
```