Automated Language Detection

Daniel Weronski Falcó Walter J. Troiani Marc Parcerisa
Facultat d'Informàtica de Barcelona (FIB), UPC, 08034 Barcelona, Spain

***Abstract*—An empirical analysis of automated language detection methodologies is presented, with particular emphasis on preprocessing techniques and their impact upon classification performance. A comprehensive investigation of tokenisation strategies — including character-level, word-level, bigram, and hybrid approaches — was conducted to determine their respective efficacies in distinguishing between languages. A dataset encompassing multiple languages was systematically processed through various preprocessing pipelines, tokenised at different granularities, vectorised using frequency-based representations, and classified using an array of machine learning algorithms. Experimental results demonstrate that bigram tokenisation coupled with minimal preprocessing and Multi-Layer Perceptron classifiers achieved the best performance, attaining an F1 score of 0.98. It was observed that increasing vocabulary size from 2000 to 5000 tokens significantly enhanced detection accuracy, whilst simultaneously diminishing the necessity for extensive preprocessing. The relationship between tokenisation strategy and classifier selection was thoroughly examined, revealing that certain preprocessing steps — particularly diacritic removal — significantly hindered classification performance for languages with these distinctive orthographic features. These findings contribute to the development of more robust language detection systems for multilingual contexts and provide insights into the statistical patterns that facilitate effective language identification.**

I. Introduction

LANGUAGE detection is a foundational challenge in natural language processing (NLP) that involves automatically identifying the language in which a text is written. As digital communication continues to expand across linguistic boundaries [1], robust language detection systems have become increasingly essential for applications ranging from machine translation to content filtering and multilingual information retrieval [2].

This report presents a comprehensive analysis of language detection methodologies, with a particular focus on the impact of preprocessing techniques and classifier selection on detection accuracy. The investigation centres on understanding how different tokenisation strategies — specifically character-level and word-level approaches —affect the model's ability to distinguish between languages, as well as how various machine learning algorithms perform when applied to this classification task.

The core objective of this work is not simply to maximise performance metrics, but rather to gain deeper insights into the linguistic and statistical patterns that enable effective language differentiation. By systematically varying vocabulary sizes, tokenisation granularity, and classification algorithms, the aim is to elucidate the underlying mechanisms that drive successful language detection and identify the trade-offs inherent in different methodological choices.

The analysis follows the standard machine learning pipeline, beginning with text preprocessing, followed by tokenisation, then feature extraction using count-based vectorisation, model training with multiple classifiers, and rigorous evaluation on a held-out test set. Throughout this process, special attention is paid to how different languages relate to one another in feature space, examining error patterns and their relationship to linguistic families, scripts, and other linguistic properties.

The findings of this investigation contribute not only to the practical implementation of language detection systems but also to a deeper theoretical understanding of how statistical and machine learning approaches can capture the distinctive characteristics of human languages from surface-level textual features.

II. Background

Language detection is one of the fundamental text classification problems in the field of Natural Language Processing (NLP). Its objective is to detect the language some input text is written in. A successful implementation may have features from intersecting fields such as linguistics, statistics and machine learning to combine their individual strengths.

Text classification problems are about analysing corpora of text in an attempt to assign predefined categories of text based on their content or features. Language detection is a specific type of classification problem wherein the languages are the categories themselves. Furthermore language detection is unique when compared to other text classification problems — *e.g.* sentiment analysis or topic categorisation — as these tend to have a heavier emphasis on content semantics. In contrast, language detection uses statistical patterns and distributions of words or characters of the text in question as the defining characteristics of the individual languages (categories).

A typical language detection process follows a pipeline such as the following [3].

A. Preprocessing

Preprocessing steps consist of applying modifications to the input data prior to any further analysis. As a general consensus, these steps result in a new text more manageable by algorithmic analyses, the general structure and meaning of which do not differ from the original's. Examples of these kinds of modifications are: Converting upper-case letters to lower-case, removing symbols that do not convey meaning, removing stop-words, lemmatizing words, etc. Huge emphasis is given to both preprocessing and the subsequent tokenisation due to the high impact these 2 tasks have on NLP tasks in comparison to choosing and training the best model.

B. *Tokenisation*

Tokenisation is the process of breaking down the text into smaller units that can be found many times throughout the data set, defining the "building blocks" (Tokens) from which the analyses will interpret the text. The key to a well-formed tokeniser is the frequency with which each of these blocks appear in the corpora. Examples of Tokenisation strategies are, from more to less granular: character-based Tokenisation strategies, which break text into its atomic particles, letters; sub-word tokenisers, such as bi-grams, tri-grams or more complex ones with variable-length tokens [4], [5]; and word-level tokenisers, which separate text into words, a task that has been shown to be tricky, as some languages do not have straight-forward word boundaries: — *e.g* Chinese, Japanese... [6].

C. *Vectorisation*

A vectorisation function takes in an ordered list of tokens and converts them into a machine-friendly numerical vector, the strategy for which entirely depends on the end goal. What distinguishes language detection from other classification problems is commonly the use of surface-level language features rather than a deeper semantic understanding. For instance, it's common to digest entire sentences into a single feature vector, counting the repetitions of a subset of all the tokens in the training data set, known as the vocabulary [7].

D. *Model fitting*

Finally, a model is fitted from the vectorised training data set, which makes the high-level decisions. This step can also come with its own challenges for the model needs to generalise and be able to predict the language of text outside the training dataset. Examples of models that have been extensively used throughout the experimentation are: Naive Bayes classifiers, Random Forests, Multi-Layer Perceptrons (MLPs), among others.

III. METHOD

The work presented aims to explore a number of approaches to language detection. Rather than testing isolated techniques, a framework with interchangeable middle steps was designed. This allows for a flexible and systematic evaluation of all the different proposed processing pipelines.

A comprehensive grid-search-based approach over all possible combinations of this problem actions and hyperparameters was performed. The most effective configurations based on the weighted F1 statistic and corpus coverage ratio were identified. The

The search space consists of dimensions that can be separated into two kinds: Preprocessing pipeline actions (Whether to take an action (URL removal, lemmatisation..) or not, similarly to Markov-Decision Process agents), and hyperparameters. The former consists of four dimensions that account for all possible combinations of preprocessor, tokeniser, vectoriser, and classifier, whilst the latter accounts for formal hyperparameters such as vocabulary size.

A. *Preprocessor*

A preprocessing class was defined to facilitate precise control over the text processing operations performed on the corpus. The steps are as follows:

- Diacritics removal whereby accents were removed.
- URL Removal, using simple regular expressions.
- Symbol Removal, using simple regular expressions.
- Sentence Splitting, whereby longer texts were divided into smaller entries within the datasets. While this modification would normally be a performed using language-dependent algorithms, this had to be simplified due to the constraints of language detection by breaking sentences on the full-stop character (".").
- Character lowercasing, which converts all uppercase characters into their lowercase equivalents.

Lemmatisation and stemming techniques were not performed as significantly modifying the words would likely compromise language detection performance. Additionally, these techniques require language-specific implementations, thereby contradicting the language-agnostic approach central to this study.

B. *Tokeniser*

Four tokenisers were defined and coded to be completely interchangeable. These are: a word, character, bigram, and a hybrid word-and-character tokeniser.

1) Word Tokenisation: Word-level tokenisation breaks down text into individual words. For instance, the sentence "Lorem ipsum dolor sit amet" would be broken down into the following list of five tokens: `['Lorem', 'ipsum', 'dolor', 'sit', 'amet']`. This approach is particularly interesting due to its ability to extract some lexical features from the languages. It operates under the assumption that different languages have different vocabularies. For example, it can capture language-specific words, such as "the" in English or "el" in Spanish.

However, this approach introduces some difficulties and shortcomings. Firstly, the number of distinct words in all languages is far larger than any realistic vocabulary size that could be set (see Section V). Secondly, some languages do not follow the common word spacing rules that other languages do (*e.g.* Chinese or Japanese). Furthermore, some languages use compound words that would be impossible to separate by a simple language-agnostic tokeniser (*e.g.* Swedish).

2) Character Tokenisation: Character-level tokenisation, (*i.e.* unigram tokenisation), breaks down text into individual characters or character sequences. The same phrase "Lorem ipsum dolor sit amet" would instead be broken down into: `['l', 'o', 'r', 'e', 'm', 'i', 'p', 's', 'u', 'm', ...]`. Depending on the task, white spaces may also be added.

Compared to word-level tokenisation, this approach captures phonological and orthographic patterns distinctive of different languages. Character-level analysis is particularly strong at capturing difference between languages with script differences (*e.g.* Latin vs Cyrillic) and frequent character or character combination patterns (*e.g.* "th" in English).

Another advantage is its ease of achieving high coverage, even with a smaller corpus of text. This is because the set of possible characters is much smaller than the set of possible words across multiple languages, meaning that character-level analysis handles unknown words more robustly.

This tokenisation strategy is far weaker for semantic analysis tasks, as each of the tokens convey no meaning by themselves. However, this shouldn't affect language detection tasks.

3) Bigram Tokenisation: Bigram tokenisation extends the character-level approach by capturing sequences of two consecutive characters within a given text. Applying this technique to the phrase "Lorem ipsum dolor sit amet" would yield the following list of overlapping bigrams: `['Lo', 'or', 're', 'em', 'm ', ' i', 'ip', 'ps', 'su', 'um', ...]`. This method ensures that each token contains some contextual relationship with its preceding and succeeding characters.

One of the key advantages of bigram tokenisation is its ability to capture short contextual dependencies while maintaining a relatively small vocabulary size compared to word-level tokenisation. This makes it particularly effective for distinguishing languages based on characteristic letter combinations. For example, the bigram "th" is highly frequent in English, while "qu" is more distinctive of French and Spanish.

Furthermore, bigram tokenisation is robust for dealing with unseen words. While word-level tokenisation may struggle with out-of-vocabulary words, and character-level tokenisation loses all semantic grouping, bigrams strike a balance by preserving partial morphological structure. This can be particularly useful in handling languages with inflections or compound words, such as German and Finnish.

Despite its advantages, bigram tokenisation has some drawbacks. The number of unique bigrams can still grow substantially when processing multilingual datasets, increasing memory and computational costs. However, for language detection, bigrams provide a useful trade-off for capturing linguistic patterns and maintaining a manageable vocabulary size.

4) Hybrid Tokenisation: Hybrid tokenisation aims to combine the strengths of word-level and character-level tokenisation. This is done by primarily segmenting text into words but further breaking down particularly long words into individual characters. For instance, given the sentence "I live in Stockholm's central station", a hybrid tokeniser would produce:

`['I', 'live', 'in', 'S', 't', 'o', 'c', 'k', 'h', 'o', 'l', 'm', 's', 'c', 'e', 'n', 't', 'r', 'a', 'l', 's', 't', 'a', 't', 'i', 'o', 'n']`

This approach is particularly useful for handling languages that either lack explicit word boundaries (*e.g.* Chinese or Japanese) or frequently use long compound words (*e.g.* Swedish or German). Instead of treating these long words as single opaque tokens, hybrid tokenisation ensures that meaningful sub-components remain accessible for analysis.

One major advantage of this method is that it retains the interpretability of word-level tokenisation while improving coverage for languages with complex word structures. By breaking down only exceptionally long words, it reduces the risk of an overly large vocabulary while still allowing character-level granularity for outlier cases. This can improve performance in multilingual settings where tokenisation strategies must be flexible.

However, hybrid tokenisation introduces some challenges. Defining an optimal threshold for "long words" is non-trivial, as different languages have varying average word lengths. Additionally, the fragmentation of long words into characters may disrupt certain linguistic features, particularly in languages where meaning is derived from full-word morphology rather than individual characters. Nonetheless, hybrid tokenisation provides an effective balance between precision and generalisation for language detection; character sequences are more often more indicative than full words.

C. Vectorisation

Language detection is commonly performed by statistically comparing token distributions in all languages in the corpus [7]. Thus, method was chosen, via `scikit-learn`'s `CountVectoriser`, which implements a Bag-of-Words model. On a high level, this tool does the following:

When fitting:

1) Creates a list of all unique tokens in the training corpus
2) Counts the frequency of each of the vocabulary tokens in the training corpus
3) Chooses the `max_features` tokens with the highest frequencies to be its vocabulary

When transforming:

1) For each document in the dataset, it counts the appearances of each of the tokens in its vocabulary.
2) Represents each of the documents as a sparse vector where each datum corresponds to the number of appearances of each of the tokens from the vocabulary in the document.

`CountVectorizer` offers a big range of features, such as tokenisation (parameter `analyzer` allows it to tokenize either by word or by character). However, due to the nature of this study, it was used solely as an engine for counting token frequencies, without any further processing steps.

D. Model Fitting

To evaluate the effectiveness of different classification techniques, multiple machine learning models were trained and tested using `scikit-learn`. These are as follows:

- **Linear Discriminant Analysis (LDA):** A linear classifier that projects the feature space into a lower-dimensional subspace while maximizing class separability, making it well-suited for structured feature distributions.
- **Decision Tree Classifier:** A non-linear rule-based model that recursively splits features to make decisions, offering interpretability but prone to overfitting on small datasets.
- **Random Forest Classifier:** An ensemble-based decision tree model that reduces overfitting by averaging multiple decision tree predictions, providing robustness to outliers and noise in the data.
- **Logistic Regression:** A simple yet effective linear model that estimates class probabilities and is particularly strong for linearly separable data.
- **Multinomial Naive Bayes (NB):** A simple probabilistic model based on Bayes' theorem [8] that assumes feature independence and I.I.D data, often performing well with sparse, high-dimensional data like text-based token counts.
- **K-Nearest Neighbors (KNN):** A non-parametric model that classifies based on proximity to labeled examples, making it useful for capturing local feature similarities.
- **Multi-Layer Perceptron (MLP) Classifier:** A feedforward artificial neural network that used a final softmax layer capable of learning complex classification decision boundaries through non-linearity and backpropagation.
- **Support Vector Classifier (SVC):** A margin-based classifier that finds an optimal hyperplane to separate classes, effective for both linear and non-linear feature spaces.

IV. Results and Discussion

A full grid search has been performed for each combination of the parameters described above, which produced a new `reports` dataset. In it, all the input parameters are recorded for each run of the script, alongside many of the performance metrics that indicate how well each combination works for the task at hand. The metrics are as follows:

1) Training data set coverage
2) Test data set coverage
3) F1 micro statistic
4) F1 macro statistic
5) F1 weighted statistic
6) Fraction of explained variance of the first PCA dimension
7) Duration of the run.

Overall, about 4000 executions of the code was performed. See [9] for the full `reports` dataset.

The first few thousand runs were "dry runs". The vocabulary size was set at 2000, allowing the search algorithm to perform as many runs as it could. In the resulting reports, the `f1_weighted` variable was taken as the target performance indicator, delivering interesting results.

A. Effects of preprocessing on weighted F1 statistic

The preprocessors were separated into families and their weighted F1 statistic was compared yielding the following:

1) Diacritics Removal: As seen in Fig. 1, removing accents and other diacritics seems to very significantly affect the results for language prediction, lowering the F1 statistic. This is likely due to the loss of precision in languages that tend to use this feature frequently over those that don't. For instance, Slavic languages make heavy use of diacritics, such as Slovak or Czech, which without them are often mistaken with Polish.

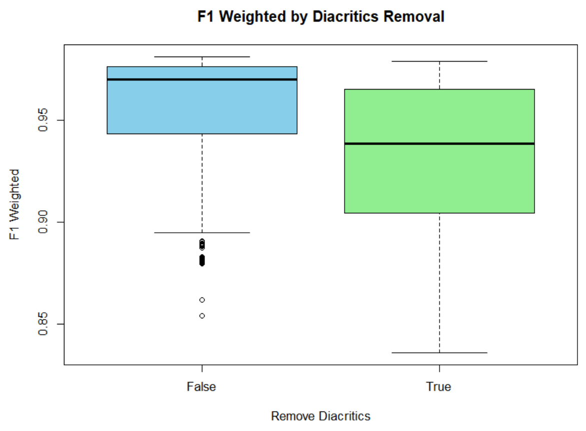

Fig. 1. Comparison of weighted F1 statistic distributions between preprocessors with and without diacritics removal. Removing diacritics shows significantly reduced classification performance.

2) URL Removal: In contrast to the removal of diacritics, removing URLs showed negligible effect, as shown in Fig. 2. This is due to the nature of the available dataset — it is not internet-related, or obtained from any other context in which URLs may be frequent. There was only one entry in the dataset containing a URL. Nonetheless, their removal was kept in the pre-processing for the sake of completeness, as these are language-independent features and provide no information for language detection.

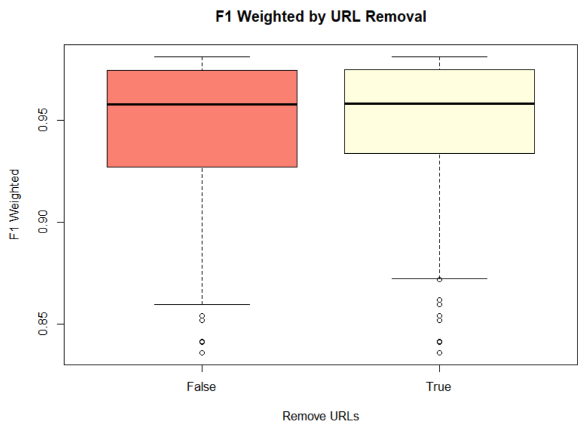

Fig. 2. Impact of URL removal on weighted F1 statistic performance. The similar distributions demonstrate negligible effect on language detection, as expected given the limited presence of URLs in the dataset.

3) Symbol Removal: In this step, symbols such as "`@`" and "`&`" are removed. This step had an effect on language detection (with a p-value of 0.02). This can be seen in Fig. 3.

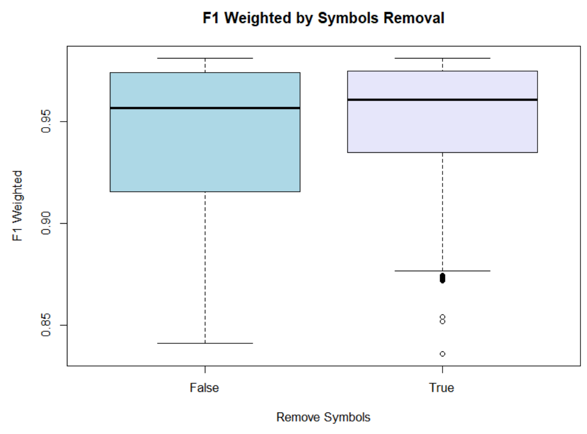

Fig. 3. Effect of symbol removal on weighted F1 classification performance. Symbol removal shows a statistically significant effect ($p = 0.02$) on language detection accuracy, suggesting certain symbols may contain language-distinguishing information.

4) Sentence Splitting: The act of splitting sentences created a larger training dataset, while reducing the average size of each example. As mentioned in section II, due to the language-agnostic language of the preprocessing step, no complex sentence splitting algorithms could be used. This means the results obtained may not be entirely representative of those obtained using more powerful algorithms. However, there is a trend whereby increasing the size of the dataset at the expense of the number of cases shows to have a negative effect in language detection, as seen in Fig. 4.

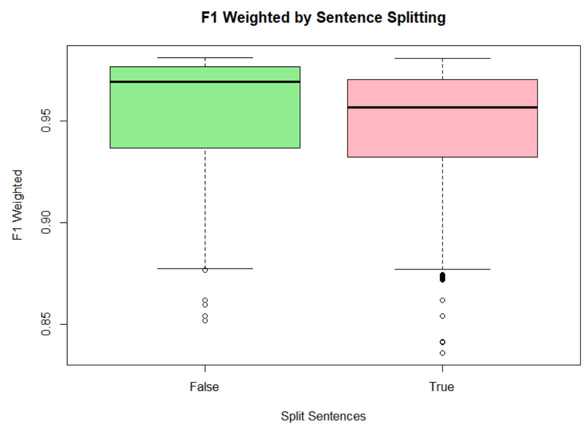

Fig. 4. Influence of sentence splitting on weighted F1 scores. Despite creating more training examples, splitting sentences yields lower performance, indicating that longer text samples provide more reliable language identification features.

5) Lowercasing: Converting all uppercase characters into lowercase proved to have a lesser effect than expected. While the F1 results increases, it does so subtly as seen in Fig. 5. This is most likely due to the tokenisers used not being overly case-sensitive. On a small dataset like the one available, the frequencies of words containing an uppercase letter are mostly 0. Furthermore, the amount of uppercase characters is negligible (relevant for character-based tokenisation).

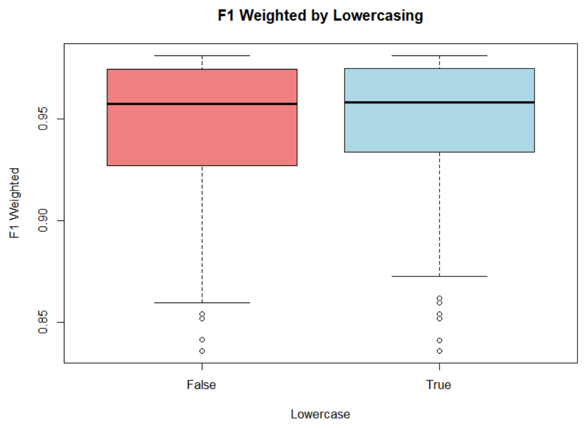

Fig. 5. Comparison of weighted F1 statistic between preprocessors with and without lowercase conversion. The modest improvement suggests that case information provides limited discriminative value for language detection tasks.

B. Effects of tokeniser in weighted F1 statistic

Tokenisers are the variable that demonstrated the largest effect in terms of language detection metrics. As expected, bigram tokenisers displayed the highest overall median performance.

Unexpectedly however, the hybrid tokeniser algorithm described in section II, showed to be the that most consistelty yielded the best results in the training dataset as shown in Fig. 6.

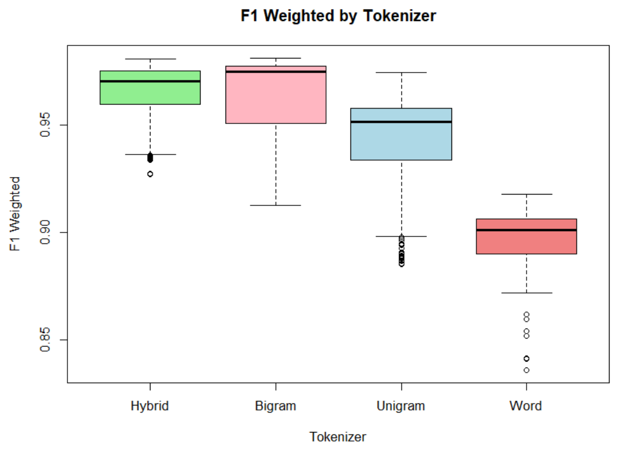

Fig. 6. Performance comparison across four tokenisation strategies. Bigram and hybrid tokenisers significantly outperform unigram and word-based approaches, with hybrid tokenisation showing the most consistent results across different classification methods.

As expected, word-based tokenisers fell short in terms of performance when compared to other methods. Much of the variability left in the tokeniser families is caused by the classifier algorithms used, explored in the following subsection.

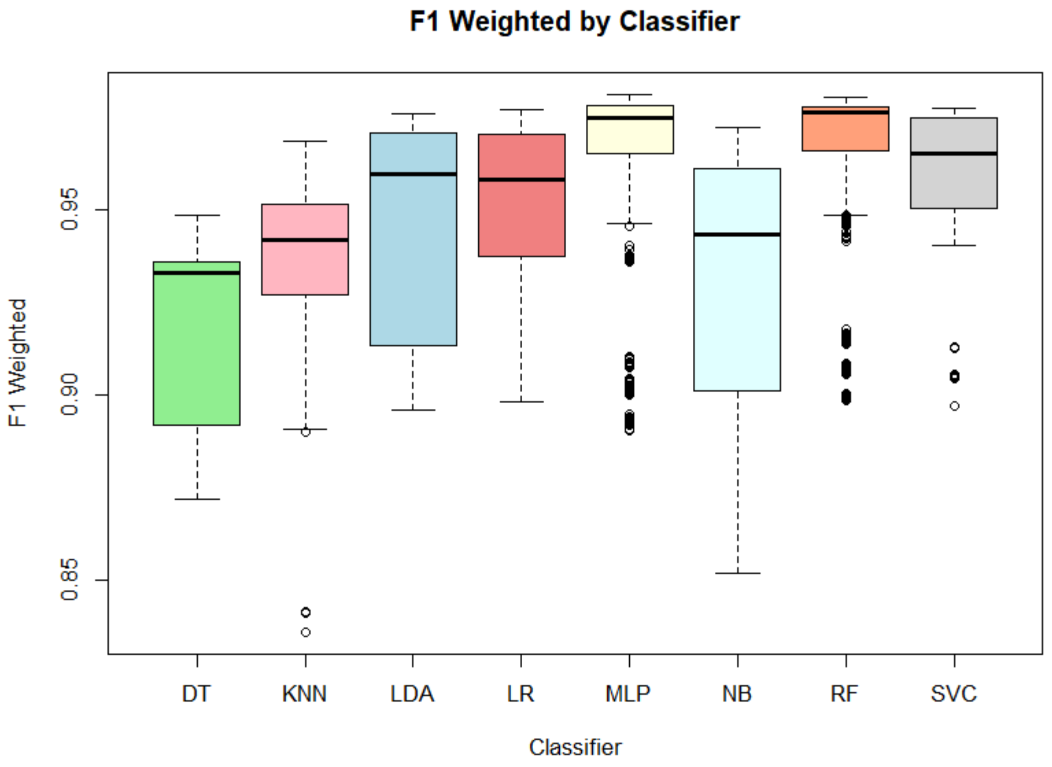

Fig. 7. Weighted F1 scores across eight classifier algorithms. Multi-Layer Perceptron (MLP) and Random Forest (RF) classifiers consistently deliver superior performance for language detection tasks compared to other approaches.

C. *Effects of classifier in weighted F1 statistic*

Choosing the best classifier proved challenging. Eight different classifiers were tested, each yielding different F1 metrics with the same parameters. As seen in Fig. 7, the two most consistently good-performing classifiers are MLPs and RF. Nonetheless, when choosing between the two the computational cost involved plays a key role in their selection. As seen in Fig. 8 the time taken for fitting an RF increases very slowly with vocabulary size, whilst MLPs rapidly grow taking up to several minutes to fit.

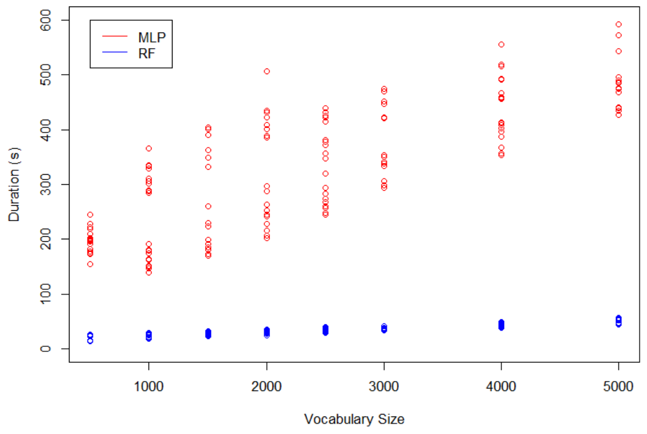

Fig. 8. Execution time comparison between MLP and RF classifiers across increasing vocabulary sizes. While both achieve similar accuracy, RF maintains consistent processing speed as vocabulary grows, while MLP computation time increases substantially.

D. *Effects of the tokeniser-classifier interaction in weighted F1 statistic*

Finally, a detailed comparison of the weighted F1 performances of all explored tokeniser-classifier combinations was carried out. As seen in Fig. 9, the bigram and hybrid (short-word) tokenisers, and MLP and RFs are the more reliably performant models.

E. *Best pipeline combination*

After identifying the most promising tokeniser-classifier combinations through initial experimentation, a more refined grid search was conducted. This search systematically varied the vocabulary size whilst operating within reasonable time and resource constraints.

For preprocessing, various combinations were tested, including URL removal, symbol removal, and character lowercasing. The tokenisation phase was restricted to bigram and hybrid approaches. The classifier evaluation was narrowed to MLP and RF algorithms, which had previously shown the most potential.

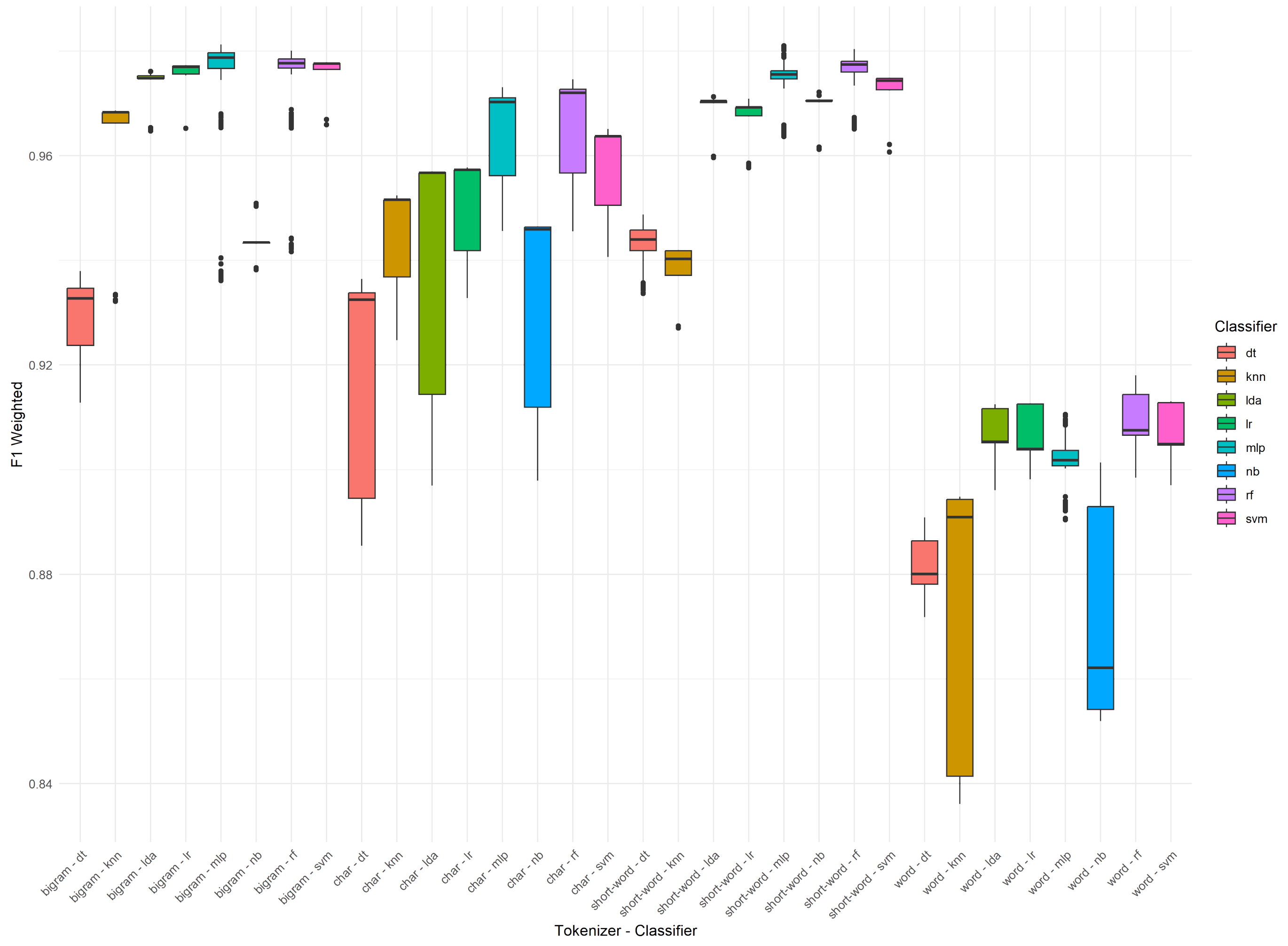

Fig. 9. Comparative performance of tokeniser-classifier combinations with a vocabulary size of 2000 tokens. Bigram and hybrid tokenisers paired with MLP and RF classifiers yield the highest and most consistent F1 scores, with specific combinations highlighted for optimal language detection.

This methodical approach enabled the determination of optimal parameter combinations for the language detection task. The analysis has been organised into two distinct components: firstly, identifying the best-performing parameter set for a fixed vocabulary size, and secondly, determining the overall optimal configuration when vocabulary size is allowed to vary.

1) Setting a Vocabulary Size of 2000: When the vocabulary size was fixed at 2000 tokens, the configuration producing the highest weighted F1 metric employed a bigram tokeniser combined with an MLP classifier. This optimal configuration included URL removal and text lowercasing, whilst sentence splitting was also applied — interestingly, despite sentence splitting generally not yielding favourable results in preliminary experimentation. Diacritic and symbol removal were not implemented in this configuration.

This parameter combination achieved the following metrics: a test coverage of 84.18%, with F1 scores of 0.9807 (micro), 0.9808 (macro), and 0.9808 (weighted).

2) Overall Best Performance: When expanding the parameter search to include larger vocabulary sizes, the highest performance was achieved with a vocabulary size of 5000 tokens (the maximum value tested in our experiments). This optimal configuration also employed a bigram tokeniser with an MLP classifier. Notably, this best-performing configuration did not implement any preprocessing steps — neither diacritic removal, URL removal, symbol removal, sentence splitting, nor lowercasing were applied.

The performance metrics for this configuration were exceptional: a test coverage of 91.73%, with F1 scores of 0.9811 (micro), 0.9812 (macro), and 0.9813 (weighted).

This finding suggests a significant insight: as vocabulary size increases, the need for preprocessing seems to decrease. The advantages that might otherwise be gained through careful preprocessing appear to be offset by the increased parameter space of larger models, which can effectively learn relevant patterns directly from the unprocessed text.

V. Conclusions

In this study, various preprocessing, tokenisation and classifier strategies for automating language detection were explored, and their impact on model performance was assessed.

The following conclusions can be drawn from the extensive experimentation conducted:

The impact of preprocessing on language detection performance was found to be significant. Actions such as the removal of diacritics, URLs, and symbols were observed to have a detrimental effect on model performance. This was particularly pronounced with diacritics, as certain languages rely heavily on these features for distinction. For instance, Slavic languages such as Czech and Slovak, which make extensive use of diacritics, were frequently misclassified as Polish when these features were removed. Sentence splitting was also noted to yield no significant improvements and, in fact, demonstrated a negative impact on the results. The minimal effect of lowercasing further emphasised the robustness of the tokenisers utilised in this investigation.

With regard to tokenisation strategies, bigram tokenisation and short-word hybrid tokenisation were demonstrated to significantly outperform word-based tokenisers in language detection tasks. Hybrid tokenisation, in particular, exhibited unexpectedly strong results with lower computational costs compared to bigram methods. This indicates that more nuanced approaches that combine multiple tokenisation techniques can lead to enhanced language identification performance.

Amongst the classifier algorithms evaluated, Multi-Layer Perceptrons (MLP) and Random Forests (RF) were found to deliver the most consistent and accurate results. This superior performance can be attributed to their capacity to capture non-linearities in the data. However, it was observed that MLPs require significantly longer longer times than RFs, despite achieving similar levels of language detection accuracy. This computational consideration makes RFs preferable, particularly in scenarios where processing efficiency is paramount.

The vocabulary size was also identified as a critical factor in language detection performance. It was found that as the vocabulary size was increased to the set limit of 5000 tokens, the need for extensive preprocessing diminished considerably. This observation suggests that larger models with more extensive feature spaces can effectively mitigate the challenges presented by unprocessed or noisy data inputs.

The optimal pipeline configuration was identified through systematic experimentation. For a fixed vocabulary size of 2000, the most effective approach involved bigram tokenisation with an MLP classifier, complemented by specific preprocessing steps including URL removal and sentence splitting. When vocabulary size constraints were removed, the best performance was achieved without any preprocessing whatsoever, further substantiating the finding that larger models and vocabulary sizes can compensate for the absence of preprocessing.

In conclusion, this study demonstrates that hybrid and bigram tokenisation strategies, when combined with MLP or RF classifiers, provide highly effective solutions for language detection challenges. The findings also highlight an important trade-off: whilst preprocessing techniques can enhance performance in certain contexts, their impact diminishes as vocabulary size increases. This finding highlights the balance that must be struck between data preparation and model complexity. The methodologies and findings presented in this research offer a foundation for the development of more accurate and adaptable language detection systems for multilingual environments.

VI. Future Work

This study has shown valuable results for language detection, but some limitations and potential improvements were identified.

The choice of weighted F1-score [10] as an evaluation metric may be reconsidered. Since the dataset was perfectly balanced across languages, F1-micro or F1-macro metrics might provide more accurate model evaluation measurements.

A key limitation in this analysis was the small size of the dataset, which limited the training potential of larger models like MLPs and restricted language detection to only a few languages. Future work should test these approaches on larger datasets with more languages.

The model performance was tested on a controlled dataset. Future research should test these models on more realistic data that includes noise and unusual cases, especially from web and social media sources, where URLs, hashtags, and symbols are common.

While this study showed good results with bigram and hybrid tokenisers, future work could test more advanced character-level tokenisation methods. Techniques such as Byte Pair Encoding (BPE) or WordPiece tokenisation might improve language detection for languages with complex word structures.

Traditional machine learning models like Random Forest and MLP showed good performance, but more advanced deep learning models could be tested. RNNs [11] and Transformers [12] might achieve better results by capturing context or long-range dependencies, which could be useful for language detection in texts that mix multiple languages.

Finally, the dataset used in this study had perfectly balanced language distribution, which is rare in real applications. Future research should examine how unbalanced language distributions affect model performance and develop methods to maintain accuracy across both common and uncommon languages.

These improvements would help create better language detection systems that work effectively across many different language situations.

APPENDIX A
PREPROCESSOR

```python
def apply(self, sentences: Iterable[str], labels: Iterable[str]) -> tuple[list[str],
    list[str]]:
    """
    Given a list of sentences, apply all the required preprocessing steps
    to clean them, and transform them into something our vectorizer + classifier
    can work with.

    This object performs steps such as sentence splitting, URL removal,
    stopword removal, lemmatization, stemming, and more.

    Args
    ----
    sentences: Iterable[str]
        Input sentences to preprocess
    labels: Iterable[str]
        Input labels to preprocess

    Returns
    -------
    tuple[list[str], list[str]]
        A tuple containing the preprocessed sentences and labels
    """
    x = sentences
    y = labels

    if self._remove_urls:  # Step 1: Perform URL Regex matching removal
        x = map(self.remove_urls, x)

    if self._remove_symbols:  # Step 2: Perform Number Regex matching removal
        x = map(self.remove_numbers_and_symbols, x)

    if self._remove_diacritics:  # Step 3: Remove diacritics
        x = map(self.remove_diacritics, x)

    if self._split_sentences:  # Step 4: Perform sentence splitting
        _x = map(self.split_sentences, x)
        x, y = self._flatten_sentences(_x, y)

    if self._lower:  # Step 5: Remove capitalization
        x = map(lambda s: s.lower(), x)

    if self._requires_tokenization():
        tokens = map(self.split_words, x)

        if self._lemmatize:  # Step 7: Lematization (optional)
            tokens = map(self.lemmatize, tokens)

        if self._stemmatize:  # Step 8: Stemming (optional)
            tokens = map(self.stem, tokens)

        x = map(" ".join, tokens)

    return list(x), list(y)
```

Listing 1. Preprocessing Functions

APPENDIX B
PREPROCESSING METHODS (IMPLEMENTATION)

```python
@staticmethod
def remove_urls(text: str) -> str:
    # Matches HTTP(S) and WWW URLs
    return re.sub(Preprocessor.URLS_PATTERN, "", text)

@staticmethod
def remove_diacritics(text: str) -> str:
    # Normalize and remove diacritical marks
    nfkd_form = unicodedata.normalize("NFKD", text)
    return "".join([c for c in nfkd_form if not unicodedata.combining(c)])

@staticmethod
def remove_numbers_and_symbols(text: str) -> str:
    return re.sub(Preprocessor.SYMBOLS_PATTERN, "", text)

@staticmethod
def split_sentences(text: str) -> list[str]:
    """
    Given a text, split it into sentences.
    """
    # In the real world, we would be using Punkt for this, but because
    # we don't know the language at this point, we'll simply split by
    # periods.
    return text.split(".")

@staticmethod
def lemmatize(tokens: Iterable[str]) -> list[str]:
    lemmatizer = WordNetLemmatizer()
    return list(map(lemmatizer.lemmatize, tokens))

@staticmethod
def stem(tokens: Iterable[str]) -> list[str]:
    stemmer = PorterStemmer()
    return list(map(stemmer.stem, tokens))
```

Listing 2. Key Preprocessing Methods

APPENDIX C
CLASSIFIER CLASS

```python
from abc import ABC, abstractmethod

import numpy as np
from numpy.typing import NDArray

# Abstract base class defining the interface for all classifiers
class Classifier(ABC):
    def __init__(self):
        pass

    @abstractmethod
    def fit(self, data: NDArray[np.float32], labels: list[str]) -> None:
        pass

    @abstractmethod
    def predict(self, data: NDArray[np.float32]) -> list[str]:
        pass

from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier

__all__ = [
    "LinearDiscriminantAnalysis",
    "RandomForestClassifier",
    "LogisticRegression",
    "MultinomialNB",
    "KNeighborsClassifier",
    "MLPClassifier",
    "SVC",
    "DecisionTreeClassifier",
]

_MAP = {
    "dt": DecisionTreeClassifier,
    "knn": KNeighborsClassifier,
    "lda": LinearDiscriminantAnalysis,
    "lr": LogisticRegression,
    "mlp": MLPClassifier,
    "nb": MultinomialNB,
    "rf": RandomForestClassifier,
    "svm": SVC,
}

options = list(_MAP.keys())

def getClassifier(name: str) -> Classifier:
    if name.lower() in _MAP:
        return _MAP[name.lower()]()
    else:
        raise ValueError(f"Invalid classifier name: {name}")
```

Listing 3. Classifier Interface, Imports, and Factory Method

References

[1] S. Dovchin, Ed., *Digital Communication, Linguistic Diversity and Education*. London: Routledge, 2020.

[2] Y. Tsvetkov and L. Derczynski, "Language detection in the age of social media," in *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2015, pp. 1529–1534.

[3] M. Lui and T. Baldwin, "A survey of language identification," *Journal of Artificial Intelligence Research*, vol. 55, pp. 143–181, 2016.

[4] T. Mikolov *et al.*, "Efficient estimation of word representations in vector space," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2013.

[5] R. Sennrich, B. Haddow, and A. Birch, "Neural machine translation of rare words with subword units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL 2016)*, 2016.

[6] D. Huang and Z. Tang, "Tokenization in chinese text: Challenges and approaches," in *Proceedings of the 2019 International Conference on Chinese Computational Linguistics (ICCCL 2019)*, 2019.

[7] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[8] T. Bayes, "An essay towards solving a problem in the doctrine of chances," *Philosophical Transactions of the Royal Society of London*, vol. 53, pp. 370–418, 1763.

[9] W. Parcerisa, Troiani, "Languagedetection," 2025. [Online]. Available: https://github.com/AimbotParce/MDS-MUD-LanguageDetection

[10] D. M. Powers, "Evaluation: From precision, recall, and f-measure to roc, informedness, markedness & correlation," *Journal of Machine Learning Technologies*, 2011.

[11] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities." *Proceedings of the national academy of sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[12] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.