Analysis and Application of Text Preprocessing for Phishing Email Detection Dataset

Charles Aaron Sarmiento
ITE Elective IV, BSIT – 401I
Jose Rizal University
Mandaluyong City, Philippines
charlesaaron.sarmiento@my.jru.edu

***Abstract*— Text preprocessing is a crucial but frequently undervalued aspect of Natural Language Processing (NLP) workflows that greatly influences the performance of machine learning models. This research offers an in-depth examination of text preprocessing methods utilized on a phishing email detection dataset comprising 18,650 emails. Employing Python libraries such as NLTK and Pandas, we developed a structured preprocessing pipeline that included converting to lowercase, substituting URLs and email addresses, removing special characters, eliminating stop words, tokenizing, and lemmatizing. Every preprocessing step was assessed separately on a test case prior to implementation on the entire dataset, enabling thorough analysis of the transformations. Findings indicate that eliminating stop-words significantly contributed to data reduction, while replacing URLs and emails successfully mitigated the domain-specific noise present in email interactions. The preprocessing pipeline effectively converted unstructured, noisy raw text into a normalized, machine-readable format ideal for training models. Analysis of word count showed considerable data refinement throughout the steps, culminating in the final lemmatized output that offers standardized tokens prepared for classification algorithms. This research establishes a fundamental structure for text preprocessing in NLP applications centered on emails and showcases the significance of organized data preparation in attaining maximum model efficacy**

Keywords*— *text preprocessing, natural language processing, phishing detection, tokenization, lemmatization, stop-word removal, email classification, NLP pipeline

I. Introduction

The continued development and innovation in the field of Natural Language Processing (NLP) has highlighted the importance of text preprocessing, ensuring that these textual data can provide meaningful insights. This step in the NLP pipeline can sometimes be overlooked. However, it is said that this can have a substantial impact on the performance of machine learning algorithms, according to Siino, Tinnirello, and La Cascia [1].

In this day and age, huge amounts of unstructured data are made available across various platforms and services. These often contain unnecessary noise, inconsistencies, and irregularities, which can impact the performance of analytical models it is used in. This further highlights the importance of being able to effectively clean, normalize, and transform these data to be useful in extracting actionable insights. As such, techniques, like tokenization, lowercasing, and stop-word removal, are often used to standardize and optimize textual data, ensuring the quality of the data.

This study aims to utilize these text preprocessing techniques to process a textual dataset, preparing it for use with a learning algorithm. This work serves as a foundation for people looking to expand their knowledge and develop the skills needed to utilize NLP in data analytics. The paper makes use of functionalities available in the Pandas and Natural Language Toolkit libraries for the Python language to process a text dataset with lowercasing, Uniform Resource Location (URL) and email tokenization, special character and stop words removal, and tokenization.

II. Related Works

Existing studies have explored the impact of text preprocessing. Various tasks utilizing NLP has been shown to have their performance improve when common preprocessing techniques, like lowercasing, stop-word removal, and tokenization, are used. This section reviews some relevant studies that evaluates the effects of these preprocessing steps.

Camacho-Collados and Pilehvar [2] have experimented on the performance of neural text classifiers when fed data which have been ran through simple text preprocessing techniques. The models were trained to be utilized for sentiment analysis and text categorization. They discovered that there is variability in the performance of models when different preprocessing techniques are applied to each. Particularly, lowercasing has shown great performance metrics when compared to the other techniques, highlighting the fact that such a simple preprocessing technique can impact the performance of a model. Though, it was also noted that great care should be placed when applying lowercasing, as it may increase ambiguity similar terms.

Zhang et al. [3] have explored the impact of language specific symbol removal, alongside word segmentation and stop-word removal, on a classifier model. The conclusion they arrived to was that the systematic use of preprocessing techniques had a positive impact on the classification of Chinese short text data. This highlights, not only the need for proper preprocessing for special symbols and characters, but also the need to customize the steps to fit the data it is being utilized on.

According to Sarica and Luo [4], stop-word removal has been a standard component of text preprocessing, but is often generalized to more non-technical words. Knowing this, they rigorously defined a list of technical stop-words for the field of engineering. To analyze its impact, they conducted a case study and they found that the customized list of stop-words has shown greatly increased performance for models utilizing it. This shows the importance of removing non-essential words in the text data being analyzed, as well as fitting said list of stop-words to the jargon of the domain it is being utilized in.

Rust et al. [5] highlighted the importance of consistent tokenization when utilized in a model. They observed that incorrectly tokenized inputs of text data often lead to degraded performance of generative tasks. Meanwhile, properly configured tokenization can lead to better overall performance, as shown in the case study they conducted in the paper. This highlights the crucial step of making sure the preprocessing of the text data to be properly done and consistent, ensuring the quality of both the input going in to the model, and the results it will produce.

May, Coterrell, and Durme [6] found that lemmatization, when applied on models trained in Russian Wikipedia articles, has significantly improved interpretability. This brought them to the conclusion that lemmtization, as a preprocessing step, may benefit topic models on morphologically rich languages. This highlights the importance of this preprocessing technique as it allows an NLP model to deal with languages that have complex linguistic structure. This step simplifies nuances of the language, making it easier for the model to understand, and ultimately allow it to make better results.

III. Methodology

A. Dataset Description

In order to experiment on the different text preprocessing techniques, a Phishing Emails dataset has been gathered from Kaggle by Chakraborty [7]. The dataset contains emails from various sender, and its contents are messages that vary from being legitimate to spam. This dataset is helpful for NLP tasks, though primarily in spam detection and text classification. The dataset, saved in a comma-separated values (CSV) file, features 18,650 emails, with each labelled as a safe email or a phishing email.

The dataset consists of the following columns:

- **Email Text** – The message contents of the email.
- **Email Type** – The label specifying if the email is considered safe or phishing.
- **An unnamed numeric column** – Assumed to be the unique identifier for the email in the dataset.

B. Preprocessing Proper

1) Installing and Importing Relevant Libraries

Before preprocessing, the libraries needed to process the textual data has to be installed. For this study, the following libraries and their modules will be installed, using the environment's module installer, which is usually *pip*.

- **Natural Language Toolkit (NLTK)** – A Python library for working with textual data, like providing modules for text processing, classification, tokenization, and parsing, among other functionalities.
- **Pandas** – This Python library provides data structures and methods that allow for handling of structured data efficiently, enabling easy data analysis and manipulation.

```
from nltk.corpus import stopwords
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer
from nltk.tokenize import word_tokenize
from nltk import pos_tag
import matplotlib.pyplot as plt
import nltk
import pandas as pd
import re
```

Fig. 1. Importing of relevant library modules to the Python environment.

Once the libraries has been installed in the system, the necessary modules will be imported to the Python environment. For this study, the following modules and methods were imported:

1. Module **stopwords** – From the *nltk.corpus* module, it provides a list of common stopwords that is needed for many languages.
2. Module **wordnet** – From the *nltk.corpus* module, it provides the functionality to parts of speech information for NLP tasks.
3. Module **WordNetLemmatizer** – From the *nltk.stem* module, it exposes the class which is used as the lemmatizer for the study.
4. Method **word_tokenize()** – From the *nltk.tokenize* module, it provides a function that allows for splitting of text into individual words, allowing for the tokenization of the text data.
5. Method **pos_tags()** – From the *nltk* module, it provides the functionality to tag a list of tokens as to which part of speech it is.
6. Module **matplotlib.pyplot** – Aliased as *plt*, it exposes functionalities that allows for graphs that can be used for data visualization.
7. Module **nltk** – The main module for the Natural Language Toolkit library, providing crucial methods in establishing an environment tailored for NLP tasks.
8. Module **pandas** – Aliased as *pd*, it provides data structure classes that enables data analysis.
9. Module **re** – The Regular Expression (RegEx) library built into the Python environment.

```
nltk.download('punkt_tab')
nltk.download('stopwords')
nltk.download('averaged_perceptron_tagger_eng')
nltk.download('wordnet')
```

Fig. 2. Downloading of the stopwords corpus and the punkt_tab tokenizer.

Once the modules has been imported into the environment, additional data that is used by the NLTK library needs to be downloaded. Namely, the *punkt_tab* tokenizer, used by the *word_tokenize()* method, and the *stopwords* corpus used by the *stopwords()* method, the *averaged_perceptron_tagger_eng* for the *pos_tags()* method, and the *wordnet* corpus used by the *wordnet* module.

2) Importing and Examining the Dataset

```
file_path = "/content/Phishing_Email.csv"
df = pd.read_csv(file_path)
df.head(10)
```

Fig. 3. Importing the dataset into a Pandas DataFrame.

To be able to make use of the dataset, which is currently contained in a CSV file, the *read_csv()* method exposed by the Pandas library is utilized. This method will read the CSV file in the specified file path and store it into a Data Frame, which is a data structure provided by the library. Once the dataset has been imported, we call the *head()*, which displays the initial rows of the data frame. In this case, the first ten rows were displayed, which allows us to confirm if the dataset has been imported successfully.

```
shape = df.shape
print('Rows: ' + str(shape[0]))
print('Columns: ' + str(shape[1]))
```

```
Rows: 18650
Columns: 3
```

Fig. 4. Displaying the shape of the dataset

The shape of the dataset is then examined, displaying the amount of rows and columns in the data frame. This is made possible with the *shape* attribute of a Data Frame, which provides a tuple holding the shape of the dataset.

```
cols = df.columns.tolist()
cols[0] = 'ID'
df.columns = cols
df.head()
```

Fig. 5. Assigning a name to a previously unnamed column in the dataset

Since it was observed that the first column in the dataset was unnamed, it was renamed to "ID", which allows the program to better reference the column to a name instead of an arbitrary index number.

```
df['Initial Email Text Word Count'] = df['Email Text'].str.split().str.len()
df.head(10)
```

Fig. 6. Creating a column storing the initial word count of the Email Text column

To allow for evaluation after the preprocessing steps, a column containing the initial word count of the "Email Text" column was created. Later on, this column will be referenced and compared to analyze how much data was changed after the preprocessing steps.

```
test_case = df['Email Text'][6]
print(test_case)
```

Fig. 7. Picking out a single row to be used as a test case for preprocessing

Before proceeding with lowercasing and other preprocessing steps, a test case was picked out. This preprocessing steps will be applied to this test case first and it will display the effects of said steps to a single entry of text data, before the steps is applied to the whole dataset.

This particular row was picked because it contains the noises that would be cleared out by the preprocessing steps, allowing us to see the effects of each preprocessing step individually.

3) Lowercasing

```
test_case = test_case.lower()
print(test_case)
```

Fig. 8. Applying lowercasing to the test case

Lowercasing was first applied to the test case. After the operation is applied, the test case is displayed to the user, allowing us to confirm if the operation was applied successfully.

```
df['Email Text'] = df['Email Text'].apply(lambda x: str(x).lower())
df.head(10)
```

Fig. 9. Applying lowercasing to the whole dataset

Once lowercasing has produced its desired results, the operation is applied to the whole dataset, using a data frame's *apply()* method. This takes in a function, either a named function, or a lambda (anonymous) function and it calls the function with the contents of each row in a data frame. In this case, the *lower()* function was ran to each row in the "Email Text" column.

Since the operation does not add or remove new words to the text data, but merely transforms the existing data into a lowercased output, getting the word after this operation is considered not needed.

4) Uniform Resource Locator (URL) and Email Address Substitution

In a study by Jang, Choi, and Allan [8], they have observed that unnatural language, like URLs, email addresses, tables, and formulas, among other things, can confuse existing NLP tools. They found that removing these unnatural language features gives an improvement in document clustering. Knowing this, removing these elements in the email dataset allows for better model performance.

```
test_case = re.sub(r'https?://[^\s]+', 'URL', test_case)
print(test_case)
```

Fig. 10. Applying the substitution of URLs to the test case.

```
test_case = re.sub(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', 'EMAIL', test_case)
print(test_case)
```

Fig. 11. Applying the substitution of Email addresses to the test case.

The operation of URL and Email Address substitution is first applied to the test case, allowing us to see its results on a single entry of data. This lets us confirm if the operation will be applied to data properly.

```
df['Email Text'] = df['Email Text'].apply(lambda x: re.sub(r'https?://[^\s]+', 'URL', x))
df['Email Text'] = df['Email Text'].apply(lambda x: re.sub(r'\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Z|a-z]{2,}\b', 'EMAIL', x))

df['After URL and Email Substitution'] = df['Email Text'].str.split().str.len()

df.head(10)
```

Fig. 12. Applying the substitution to the whole dataset

Once it has been confirmed the operation produces its desired results, both substitution operations are ran to the

whole dataset. The word count of the text after the operation is saved to a new column to allow for evaluation after preprocessing is done.

5) Special Characters Removal

```
test_case = re.sub("[^a-zA-Z0-9\s]", '', test_case)
print(test_case)
```

Fig. 13. Removing special characters from the test case.

The removal of special characters is first applied to the test case, allowing us to see what characters will be left in the dataset after the operation. This is made possible using a RegEx looking for non-alphanumeric characters in the text data and omitting them. This allows us to see if the operation will produce the desired result.

```
df['Email Text'] = df['Email Text'].apply(lambda x: re.sub("[^a-zA-Z0-9\s]", '', x))
df['After Special Character Removal'] = df['Email Text'].str.split().str.len()
df.head(10)
```

Fig. 14. Removing special characters from the whole dataset.

The removal of special characters is applied to the whole dataset. After the operation, the word count is stored in a new column to allow for evaluation after preprocessing.

6) Stop-word Removal

```
stop_words = stopwords.words('english')
print(stop_words)
```

Fig. 15. Displaying the English stopwords available in the NLTK library.

Before its removal, the list of stop-words is first displayed to give us an idea of what words should be expected to be removed from the text data.

```
def remove_stopwords(text, stopword_list):
  final_text = []
  for word in text.split():
    if word == '' or '\r\n' in word or word in stopword_list:
      None
    else:
      final_text.append(word)
  return ' '.join(final_text)
```

Fig. 16. Defining a function to allow for removal of stop-words in a given text.

To enable the functionality of removing stop-words, a custom function was defined, which scans through the given text and checks if the word is included in the list of stop-words. Once the function finishes executing, it returns the new text devoid of the given stop-words.

```
remove_stopwords(test_case, stop_words)
```

Fig. 17. Removing stop-words from the test case.

```
df['Email Text'] = df['Email Text'].apply(lambda x: remove_stopwords(x, stop_words))
df['After Stopword Removal'] = df['Email Text'].str.split().str.len()
df.head(10)
```

Fig. 18. Removing stop-words for the whole dataset.

Once the function is defined, it is first ran with the test case to allow us to check if it works as intended. When this is confirmed, the function is applied to the whole dataset. As with the previous operation, the word count is saved in a new column to allow for evaluation after the preprocessing of the data.

7) Tokenization

```
print(word_tokenize(test_case))
```

Fig. 19. Applying tokenization

To enable the functionality of tokenization, the *word_tokenize()* method from the *NLTK.tokenize* module was used. This method outputs an array of all the words in the text, which can then be fed to a model for training or testing.

```
df['Tokenized Email Text'] = df['Email Text'].apply(word_tokenize)
```

Fig. 20. Applying the tokenizer to the whole dataset.

Once the tokenization operation is confirmed to produce the desired results in the test case, it is then applied to the whole dataset. Since this operation does not add or remove words in the text data, there is no need to save the word count for the data into a new column.

8) Lemmatization

```
print(pos_tag(test_case))
```

Fig. 21. Tagging for Parts of Speech for each word the test case.

Before the text can be lemmatized, it has to be tagged for what Part of Speech (POS) each word is. In this case, the *pos_tag()* method of the NLTK module is used.

```
def get_wordnet_pos(tag):
  if tag.startswith('J'):
    return wordnet.ADJ
  elif tag.startswith('V'):
    return wordnet.VERB
  elif tag.startswith('N'):
    return wordnet.NOUN
  elif tag.startswith('R'):
    return wordnet.ADV
  else:
    return wordnet.NOUN
```

Fig. 22. Defining a function to convert NLTK POS tags to WordNet POS tags.

The lemmatizer used in the study uses a different format than the one provided by the *pos_tag()* method. As such, a function had to be defined converting the default format to the WordNet format understood by the lemmatizer.

```
lemmatizer = WordNetLemmatizer()
def lemmatize_tokens(tokens):
  pos_tags = pos_tag(tokens)
  return [lemmatizer.lemmatize(word, get_wordnet_pos(pos)) for word, pos in pos_tags]
```

Fig. 23. Defining the lemmatizer used, as well as the function that will lemmatize each token in the dataset.

The lemmatizer used is the WordNetLemmatizer, which is included in the *NLTK.stem* module. This is then used in the *lemmatize_tokens()* method that will be use to apply lemmatization to a list of tokens. Inside the function, pos_tags are generated for the tokens, and a list of lemmatized tokens are generated and returned.

```
test_case = lemmatize_tokens(test_case)
print(test_case)
```

Fig. 24. Lemmatizing the test case.

Once the lemmatizer and its function is defined, we first apply it to the test case to confirm that the results are what is expected.

```
df['Lemmatized Email Text'] = df['Tokenized Email Text'].apply(lemmatize_tokens)
df['Lemmatized Email Text'].head(10)
```

Fig. 25. Applying lemmatization to the whole dataset.

When it is confirmed that the lemmatizer works as intended, the step is applied to the whole dataset. The result of the operation is stored in a new column of the data frame.

IV. Results

This section describes the outcomes of the preprocessing steps that was done on the dataset. This includes the transformations done, showing how the email text looked like before and after each step of preprocessing.

A. Before and After Preprocessing

Step	Text
Initial (before preprocessing)	On Sun, Aug 11, 2002 at 11:17:47AM +0100, wintermute mentioned: > > The impression I get from reading lkml the odd time is > > that IDE has gone downhill since Andre Hedrick was > > effectively removed as maintainer. Martin Dalecki seems > > to have been unable to further development without > > much breakage. > > Hmm... begs the question, why remove Handrick? > If it ain't broke, don't fix it. See, the IDE subsystem is like the One Ring. It's kludginess, due to having to support hundreds of dodgy chipsets & drives means that it is inherintly evil. A few months of looking at the code can turn you sour. Years of looking at it will turn you into an arsehole. They haven't found a hobbit that can code, so mortal humans have to suffice. Kate -- Irish Linux Users' Group: ilug@linux.ie http://www.linux.ie/mailman/listinfo/ilug for (un)subscription information. List maintainer: listmaster@linux.ie
After lowercasing	on sun, aug 11, 2002 at 11:17:47am +0100, wintermute mentioned: > > the impression i get from reading lkml the odd time is > > that ide has gone downhill since andre hedrick was > > effectively removed as maintainer. martin dalecki seems > > to have been unable to further development without > > much breakage. > > hmm... begs the question, why remove handrick? > if it ain't broke, don't fix it. see, the ide subsystem is like the one ring. it's kludginess, due to having to support hundreds of dodgy chipsets & drives means that it is inherintly evil. a few months of looking at the code can turn you sour. years of looking at it will turn you into an arsehole. they haven't found a hobbit that can code, so mortal humans have to suffice. kate -- irish linux users' group: ilug@linux.ie http://www.linux.ie/mailman/listinfo/ilug for (un)subscription information. list maintainer: listmaster@linux.ie
After substituting URLs and email address	on sun, aug 11, 2002 at 11:17:47am +0100, wintermute mentioned: > > the impression i get from reading lkml the odd time is > > that ide has gone downhill since andre hedrick was > > effectively removed as maintainer. martin dalecki seems > > to have been unable to further development without > > much breakage. > > hmm... begs the question, why remove handrick? > if it ain't broke, don't fix it. see, the ide subsystem is like the one ring. it's kludginess, due to having to support hundreds of dodgy chipsets & drives means that it is inherintly evil. a few months of looking at the code can turn you sour. years of looking at it will turn you into an arsehole. they haven't found a hobbit that can code, so mortal humans have to suffice. kate

	-- irish linux users' group: EMAIL URL for (un)subscription information. list maintainer: EMAIL
After removing special characters	on sun aug 11 2002 at 111747am 0100 wintermute mentioned the impression i get from reading lkml the odd time is that ide has gone downhill since andre hedrick was effectively removed as maintainer martin dalecki seems to have been unable to further development without much breakage hmm begs the question why remove handrick if it aint broke dont fix it see the ide subsystem is like the one ring its kludginess due to having to support hundreds of dodgy chipsets drives means that it is inherintly evil a few months of looking at the code can turn you sour years of looking at it will turn you into an arsehole they havent found a hobbit that can code so mortal humans have to suffice kate irish linux users group EMAIL URL for unsubscription information list maintainer EMAIL
After removing stop-words	sun aug 11 2002 111747am 0100 wintermute mentioned impression get reading lkml odd time ide gone downhill since andre hedrick effectively removed maintainer martin dalecki seems unable development without much breakage hmm begs question remove handrick aint broke dont fix see ide subsystem like one ring kludginess due support hundreds dodgy chipsets drives means inherintly evil months looking code turn sour years looking turn arsehole havent found hobbit code mortal humans suffice kate irish linux users group EMAIL URL unsubscription information list maintainer EMAIL
After tokenization	['sun', 'aug', '11', '2002', '111747am', '0100', 'wintermute', 'mentioned', 'impression', 'get', 'reading', 'lkml', 'odd', 'time', 'ide', 'gone', 'downhill', 'since', 'andre', 'hedrick', 'effectively', 'removed', 'maintainer', 'martin', 'dalecki', 'seems', 'unable', 'development', 'without', 'much', 'breakage', 'hmm', 'begs', 'question', 'remove', 'handrick', 'aint', 'broke', 'dont', 'fix', 'see', 'ide', 'subsystem', 'like', 'one', 'ring', 'kludginess', 'due', 'support', 'hundreds', 'dodgy', 'chipsets', 'drives', 'means', 'inherintly', 'evil', 'months', 'looking', 'code', 'turn', 'sour', 'years', 'looking', 'turn',
	'arsehole', 'havent', 'found', 'hobbit', 'code', 'mortal', 'humans', 'suffice', 'kate', 'irish', 'linux', 'users', 'group', 'EMAIL', 'URL', 'unsubscription', 'information', 'list', 'maintainer', 'EMAIL']
After lemmatization	['sun', 'aug', '11', '2002', '111747am', '0100', 'wintermute', 'mention', 'impression', 'get', 'read', 'lkml', 'odd', 'time', 'ide', 'go', 'downhill', 'since', 'andre', 'hedrick', 'effectively', 'remove', 'maintainer', 'martin', 'dalecki', 'seem', 'unable', 'development', 'without', 'much', 'breakage', 'hmm', 'begs', 'question', 'remove', 'handrick', 'aint', 'break', 'dont', 'fix', 'see', 'ide', 'subsystem', 'like', 'one', 'ring', 'kludginess', 'due', 'support', 'hundred', 'dodgy', 'chipsets', 'drive', 'mean', 'inherintly', 'evil', 'month', 'look', 'code', 'turn', 'sour', 'year', 'look', 'turn', 'arsehole', 'havent', 'find', 'hobbit', 'code', 'mortal', 'human', 'suffice', 'kate', 'irish', 'linux', 'user', 'group', 'EMAIL', 'URL', 'unsubscription', 'information', 'list', 'maintainer', 'EMAIL']

Fig. 26. The test case after each preprocessing step, from initial (not preprocessed) to after lemmatization.

The table displays the test case, as previously selected, and how it looks like after each preprocessing step. The column “Email Text” contains the email message body, and it gets overwritten when the text gets lowercased, have its URLs and email addresses substituted, have special characters removed, and have the stop-words removed. When the text gets tokenized, it gets stored in a new column named “Tokenized Email Text”, containing a list of words (tokens). This column gets referenced when the text undergoes the final step of preprocessing, or lemmatization. It contains simplified versions of the tokens. The result of this last operation is then stored in “Lemmatized Email Text.” The contents of this column should then be used as a training-ready dataset.

B. *Word Count Summary*

```
values = [df[col].sum() for col in cols]

colors = ['#4C72B0', '#55A868', '#C44E52', '#8172B3']

plt.figure(figsize=(12, 6))
bars = plt.bar(
    range(len(cols)),
    values,
    width=bar_width,
    tick_label=cols,
    color=colors,
    label='Word Count'
)

for i, bar in enumerate(bars):
    height = bar.get_height()
    plt.text(
        bar.get_x() + bar.get_width() / 2,
        height + max(values) * 0.01,  # offset above bar
        str(height),
        ha='center',
        va='bottom'
    )

plt.ylabel("Total Word Count")
plt.title("Total Word Count at Each Preprocessing Step")
plt.xticks(rotation=15, ha='right')
plt.legend()
plt.tight_layout()
plt.show()
```

Fig. 27. Visualization of word count after preprocessing steps using Matplotlib Bar Graph.

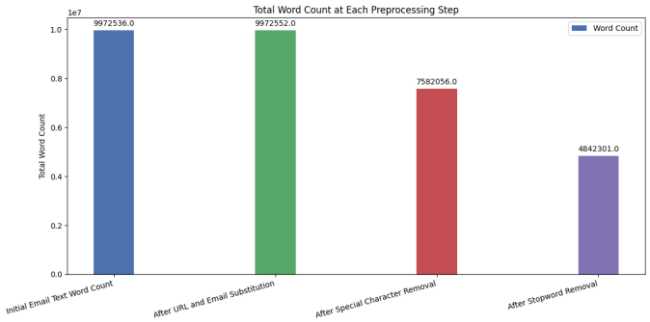

Fig. 28. Bar graph of word counts after preprocessing steps.

The graph shows the word count of the text data after each preprocessing step. It is worth noting that not all steps were displayed. Techniques like lowercasing, tokenization, and lemmatization only transform words and do not add or remove words, therefore they are chosen to be excluded from the graph. It can be said that stop-word removal has decreased word count the most out of all steps, signifying that the technique may have the most impact in a model's performance due to how much data it tends to remove from the set.

V. Discussion

A. *Interpretation of Results and Significance*

The step of preprocessing textual data, before being utilized in a model, demonstrated improvements in text clarity, structure, and interpretability. The transformations done with each step in the preprocessing pipeline has made unstructured, "messy," raw text to cleaner, more normalized data. As observed from the reduced word count, the preprocessing has been able to remove irrelevant tokens that can degrade the performance of the model that will utilize these textual data.

B. *Impact of Each Preprocessing Step*

The various preprocessing that was undergone for the text data managed to contribute to the data refinement process. Firstly, lowercasing managed to remove duplication, though care should be used, especially when dealing with proper nouns that have a similar name to a common. An example of this would be the words apple, the fruit, and Apple, the company. The removal of special characters further eliminated noise that could otherwise introduce confusion and skew results for the model. Stop-word removal, having had the most significant impact in terms of removed word count, reflects the significant amount of high-frequency terms that contribute little to no impact on a model's performance. Tokenization helped structure the data into a list that can be easily read through, and lemmatization reduced the word to their base forms (lemmas), simplifying it and ensuring consistency within the dataset. Overall, these steps helped the data be of good quality, such that it can be ready to be used in a model.

C. *New Noise Identification*

The new noise found in this dataset is the URLs and email addresses that is expected to show up in email messages. These tokens, as previously stated to be unnatural and can negatively impact a model's performance, has been removed with the use of RegEx patterns. While this step didn't amount to much movement in the dataset's word count, it did still clean the dataset from noise that could affect how well the quality of a model's results will be.

D. *Limitations and Trade-Offs*

Despite the benefits from each preprocessing steps, they do still introduce trade-offs that need to be balanced. Stop-word removals, while the most impactful, could lead to loss of context if not handled properly. There is also the fact that some fields or domains can have their own stop-words that provide little value in providing meaning, and should be put into consideration if they should be removed alongside the more general stop-words. The URL and email address substitution step could also use a more specific pattern for identifying such features, allowing for more edge cases to be covered by the step. Models trained in a preprocessing pipeline similar to the one defined in this study could benefit from more fine-tuning with the techniques used.

VI. Conclusion

This research highlighted the significance of a methodical and logical strategy for text preprocessing in NLP applications. Through the implementation of a series of widely used preprocessing methods—such as converting to lowercase, replacing emails and URLs, removing special characters and stop words, tokenizing, and lemmatizing—the dataset was successfully converted from chaotic and unstructured raw text into a more uniform and machine-friendly format. Every method aided in minimizing redundancy, enhancing interpretability, and making the data

ready for more efficient training and analysis. These results support the assertions of recent studies like May et al. [6] and Camacho-Collados and Pilehvar [2] which highlight the important impact of preprocessing on the effectiveness of NLP pipelines. Future endeavors might focus on assessing the impact of these preprocessing steps on the real classification performance of phishing detection models, including logistic regression or deep learning frameworks.

REFERENCES

[1] M. Siino, I. Tinnirello, and M. La Cascia, "Is text preprocessing still worth the time? A comparative survey on the influence of popular preprocessing methods on Transformers and traditional classifiers," Information Systems, vol. 121, p. 102342, Mar. 2024. doi:10.1016/j.is.2023.102342.

[2] J. Camacho-Collados and M. T. Pilehvar, "On the role of text preprocessing in neural network architectures: An evaluation study on text categorization and sentiment analysis," Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP, 2018. doi:10.18653/v1/w18-5406.

[3] D. Zhang, J. Li, Y. Xie, and A. Wulamu, "Research on performance variations of classifiers with the influence of pre-processing methods for Chinese short text classification," PLOS ONE, vol. 18, no. 10, Oct. 2023. doi:10.1371/journal.pone.0292582.

[4] S. Sarica and J. Luo, "Stopwords in technical language processing," PLOS ONE, vol. 16, no. 8, Aug. 2021. doi:10.1371/journal.pone.0254937.

[5] K. Sun et al., "Tokenization consistency matters for generative models on extractive NLP tasks," Findings of the Association for Computational Linguistics: EMNLP 2023, pp. 13300–13310, 2023. doi:10.18653/v1/2023.findings-emnlp.887.

[6] C. May, R. Cotterell, and B. Van Durme, "An Analysis of Lemmatization on Topic Models of Morphologically Rich Language," arXiv preprint arXiv:1608.03995, 2016. [Online]. Available: https://doi.org/10.48550/arXiv.1608.03995

[7] S. Chakraborty, "Phishing email detection," Kaggle, https://www.kaggle.com/datasets/subhajournal/phishingemails/data (accessed Aug. 6, 2025). M. Jang, J. D. Choi, and J. Allan, "Improving Document Clustering by Removing Unnatural Language," in Proceedings of the 3rd Workshop on Noisy User-generated Text (WNUT), Copenhagen, Denmark, Sept. 2017, pp. 122–130. doi: 10.18653/v1/W17-4416.

[8] M. Jang, J. D. Choi, and J. Allan, "Improving Document Clustering by Removing Unnatural Language," in Proceedings of the 3rd Workshop on Noisy User-generated Text (WNUT), Copenhagen, Denmark, Sept. 2017, pp. 122–130. doi: 10.18653/v1/W17-4416.