Module Code: CS3TM

Assignment report Title: Text Mining and NLP

Student Number: 31011732

Date Completed: 24/05/2025

Actual hours spent: 7

Data set information derived from student number: 310117329

This is how to identify which data set to use (Please copy the following information in report front page).

```
[ ] index=input('type your student number?')
```

```
type your student number?310117329
```

```
[ ] x=divmod(int(index),4)
    yourdata1=x[1]
    y=divmod(int(index),3)
    yourdata2=y[1]

    print('This is your data set index ----> (', x[1], y[1], ')' )
```

```
This is your data set index ----> ( 1 0 )
```

NOTE: If your two data sets indices are the same, please add your student number a small number, try again.

```
[ ] data1= twenty_train.target_names[x[1]]
    data2= twenty_train.target_names[y[1]]
    categories1=[data1,data2]
    print(categories1)
```

```
['comp.graphics', 'alt.atheism']
```

Abstract

This report explores the use of natural language processing and machine learning techniques. This focuses on binary classification on a subset of the 20 Newsgroups dataset, including two categories – alt.atheism and comp.graphics. By reading this report one can dive into the effects of different preprocessing methods on classification performance including text normalization, tokenization, stop word removal, stemming and lemmatization can be found. A TF-IDF weighted bag-of-words model is used for feature extraction, with classification being conducted via logistic regression within multiple scikit-learn pipelines. Results demonstrate how advanced preprocessing, mainly removal of non-informative content and lemmatization, causes improvements in accuracy and additional evaluation metric scores over the base pipeline. The findings also show that the preprocessing and normalization techniques must be suited to the dataset and text classification task, as a combination of certain techniques does not necessarily increase the model's performance.

Table of Contents

1. Introduction

Due to the sudden rapid upsurge of text data available digitally, text mining and natural language processing (NLP) have become increasingly more vital tools for obtaining meaningful information from disorganized and unstructured text data. There are a wide range of libraries available, such as NLTK that provide tools for linguistic analysis (Bird et al 2009). Text mining involves utilizing computational techniques to deconstruct and dissect huge volumes of text data in order to discover patterns, trends and insights that is substantially more difficult and time-consuming to carry out manually. NLP involves using artificial intelligence to allow devices to not only understand but interpret and generate human language.

The wide range of applications of text mining and NLP is thoroughly explored in this report for classification of newsgroup posts – specifically a set of the educational 20 Newsgroups dataset. There are two main objectives here: firstly, to analyse and derive linguistic features from text and secondly, to implement and evaluate a pipeline for machine learning for classifying newsgroup into two separate categories.

Linguistic features include: morphology – the structure and form of words, lexicon – the vocabulary and meaning of words, syntax – rules for how sentences and phrases are formed and how words are arranged to determine this and semantics – what meanings words, phrases and sentences convey. This pipeline makes use of NLP preprocessing, feature extraction via a term-frequency-inverse document frequency (TF-IDF) weighted unigram bag-of-words model as well as supervised classification through logistic regression (Bird et al 2009; Scikit-learn developers 2024).

Employing these methods systematically is this report's aim as it will demonstrate how effective NLP feature extraction and machine learning are in text classification. Furthermore, the results and discussion section will underline the various strengths and weaknesses of the approaches selected, imparting insights into the challenges and technical process in automated text analysis.

2. Methodology

In this section, a view at the concepts and methods followed in the text processing and classification pipeline is present. There are two main tasks of firstly extracting the linguistic features using NLP techniques and secondly developing and evaluating a supervised machine learning model to classify newsgroup posts.

2.1. NLP Analysis & Linguistic Feature Extraction

For extraction of meaningful features from the unstructured posts, a multitude of NLP techniques were executed to target various linguistic levels. For morphological analysis, tokenization was deployed to separate raw text into

individual words/tokens. These tokens were then processed further to their root forms through stemming and lemmatization, reducing words to a common base e.g., “walking” to “walk”. This makes word forms consistent and lowers dimensionality.

Lexical analysis was carried out via incorporating a custom text processor to clean the text by removing punctuation, special characters and most commonly found stop words in the English language. A vocabulary/lexicon was then created from the tokens that remained, building the foundation for more feature extraction. Regarding syntactic analysis, a label was applied to every token with its grammatical category using part-of-speech (POS) tagging. This gathers patterns in syntax and allows for a deeper focus on specific types of words (nouns, adjectives, verbs).

Basic semantic features were obtained through lemmatization and identifying named keywords. Whilst a deep semantic understanding was not the target, the pipeline is flexible in implementing more advanced semantic features if required. The overall code utilizes the NLTK library for all steps in preprocessing (Bird et al 2009). Custom “processText” and “cleanText” methods were developed to streamline tokenization, stop word removal and stemming/lemmatization to ensure that the text cleaning was consistent and efficient across the newsgroup posts.

2.2. Text Classification Pipeline

Regarding dataset selection, two categories were picked from the 20 Newsgroup dataset based on the student number which established a unique and reproducible subset. An additional number was added so that the data indices were not identical. Both splits for training and testing sets were gathered for these two categories. From there, a TD-IDF weighted unigram bag-of-words was adopted to be the baseline feature extraction method. This transformed the cleaned text into feature vectors to extract the importance of words within and across newsgroup posts.

To automate the sequence of operations, the scikit-learn Pipeline tool was utilized (Scikit-learn developers 2024). This allowed for text vectorization via a custom analyser, the TD-IDF transformation and classification. This formed a modular design that made sure all preprocessing and feature extraction techniques were applied consistently for both training and testing. The chosen classifier ended up being logistic regression as it is effective and interpretable in tasks involving text classification (Scikit-learn developers 2024). This model was trained on the training data and then evaluated on the test data. Due to the modular design, alternative classifiers can be swapped with linear regression to compare effectiveness between models. Non-linear classifiers were not considered as they would most likely end up being too complex for a simple dataset such as this, and the increase in computational power required would be wasted.

The evaluation assesses model performance using standard metrics for classification – accuracy, precision, recall and F1-score. A deeper understanding of the results is outlined by a confusion matrix and an analysis of

misclassified examples, clearly showcasing the strengths and weaknesses of the model. The code was implemented in the Python language using NLTK and scikit-learn libraries. The full pipeline and code setup is available through the appendix.

3. Data, Experiments & Evaluation

3.1 Task Description

The focus of this experiment is to explore the effectiveness of NLP techniques and machine learning for text classification of a subset of the 20 Newsgroup dataset (Scikit-learn developers 2024). As mentioned previously, this objective can be divided into two main tasks with the first task being processing the raw text data and gathering significant features from it (Bird et al 2009; Jurafsky & Martin 2023). The second task being using these obtained features to train and then evaluate a supervised classifier. The categories chosen based on student number are alt.atheism and comp.graphics. The dataset has been split into training and test sets, with 319 documents in the alt.athesim category and 389 in the comp.graphics category for training alongside a similar distribution for the test set.

The beginning of this experiment involves analysing the linguistic characteristics of the text data and then predicting the category of unseen documents by implementing machine learning (Bird et al 2009; Scikit-learn developers 2024). The dataset was required to be consistent and representative with minimal noise. This was accomplished by the removal of redundant text data such as headers and signatures during the preprocessing stage. There are five pipelines in total to fully explore the effectiveness of NLP techniques and its effects on machine learning. This is a key focus of this experiment and allows for a deep discussion on which preprocessing techniques are the most effective, as well as whether the classifier choice was suitable based on the results of the individual pipelines.

3.2 NLP Analysis & Preprocessing

To effectively classify texts, thorough preprocessing is required to convert raw and unstructured data into a more suitable format for feature extraction. In this experiment, a multi-stage preprocessing pipeline using NLP was employed to reduce noise and normalize the text whilst gathering meaningful features at the morphological, lexical, syntactic and minor semantic levels.

To begin, the text must be preprocessed by removing possibly misleading and non-informative, seen in Appendix Figure 3.2.3. Here the text is tokenized into separate words whilst additionally transforming all characters to lowercase and removing stop words (see Appendix Figure 3.2.4). Furthermore, stemming or lemmatization can be deployed to reduce the tokens to their root forms to group similar words and reduce feature space

dimensionality. The skeleton code does not include stemming in the “processText” method so it has been integrated here, following the lemmatization implementation that was included. Separate pipelines can be developed using this base preprocessing function that allows for opting in either lemmatization, stemming or both.

After cleaning the data, the text is then further processed via the “processText” function which strips content like email addresses, headers and URLs. As shown in Appendix Figure 3.2.1, a custom “cleanText” function was introduced which utilizes regular expressions to strip out lines that start with headers that are usually found in emails, like “From:” and “Subject:”, as well as email addresses. This allows the model to focus on the message content rather its metadata. This more thoroughly preprocesses the text than the base “processText” method provided in the skeleton code as the regex expression given does not remove full email addresses. The “@\w+” only removes an “@” followed by word characters and not full email addresses – typically username handles as used by Twitter. The only case for an email being removed by this regular expression is if the email is just “@username” which is unlikely. By looking at Appendix Figure 3.2.2, it is clear that redundant information is picked out, reducing noise and focusing the data on the content.

Finally, the syntactic structure of the text is analysed via a “getTags” function which assigns POS tags to every token, as seen in Appendix Figure 3.2.5. This delivers a view on the grammatical composition of the documents which is useful for deeper feature engineering or analysis on linguistics. See how a list of tokens and their POS tags are outputted in Appendix Figure 3.2.6. These preprocessing stages are compared systematically in the results section – “processText” is the main preprocessing with “cleanText” further processing the text data.

3.3 Logistic Regression Classification

The newsgroup posts were classified via a logistic regression model that was integrated within a modular pipeline using scikit-learn. After the thorough preprocessing and feature extraction carried out previously, every document was then rendered as a numerical feature vector through a TF-IDF weighted unigram bag-of-words. This representation of each document spotlights informative and unique words while disregarding common terms. This culminates in a sparse and highly dimensional feature space that is very appropriate for linear classifiers.

To make sure that the process is consistent and replicable, this pipeline was built to tokenize, normalize, feature extract and classify automatically. The custom “processText” method was utilized for further preprocessing to guarantee that tokenization, case normalization and removal of stop words are applied consistently to every document that is given to the classifier. Subsequently, “CountVectorizer” and “TfidfTransformer” was used to transform the processed tokens into TF-IDF feature vectors. In this experiment, there is a pipeline for just the raw text with “processText”, then one with further preprocessing using “cleanText”. Additionally, due to the modular

nature of this experiment, pipelines for lemmatization, stemming and a combination of both using the cleaned text have also been built.

The chosen classifier was logistic regression – this is due to its established efficiency and interpretability in text classification, primarily tasks that include highly dimensional feature sets created via bag-of-words models. This model was tinkered and set to have a suitable maximum iteration parameter to ensure convergence whilst training. The training took place on the labelled training set which consisted of documents from both categories with the pipeline suiting the feature extraction and classification techniques separately. These pipelines can be seen with all relevant parameter settings in Appendix Figure 3.3.1.

As the training is completed, the models are ready to predict the categories that unseen testing documents are by harnessing the same preprocessing and feature extraction steps to output the classification. Each pipeline is trained and evaluated independently (see Appendix Figures 3.3.2, 3.3.3), allowing for the performance of each pipeline's model to be evaluated and discussed in detail in the next section.

3.4 Results & Discussion

The test set created from both chosen categories from the dataset was used by the logistic regression classifier after the NLP preprocessing and feature extraction pipeline. It was crucial that the test set embraced an even representation of both categories so that the metrics reveal the generalization ability of the model across the two unique topics.

To assess the effectiveness of the NLP preprocessing, five separate variants of the classification pipeline were experimented with. Firstly, a baseline model was assessed which had the least amount of preprocessing using the raw data with only "processText". The second model used the "cleanText" method, thus using the cleaned text data rather than the less processed data. The third model utilized both clean text and lemmatization, with the fourth model using stemming instead and a final fifth model employing both lemmatization and stemming.

This approach allowed for a thoughtful evaluation of the most advanced pipeline and the prior, less advanced pipelines that all make use of different NLP preprocessing. It is predicted that the baseline model achieves effective results indicated by the metrics, with the addition of cleaned text data boosting the effectiveness. The implementation of lemmatization should prove to be more effective than stemming, with the pipeline that utilizes both barely affecting the performance. The code for generating performance metrics is shown in Figure 3.4.1. For further analysis of errors, misclassified examples combined a confusion matrix are generated, as seen in Figure 3.4.2

The baseline pipeline utilized the raw text after "processText" with just basic tokenization and stop word removal. As visualized in Appendix Figure 3.4.3 , this yielded a very good score of 95.05% accuracy with a weighted average of 0.95 precision, recall and f1-score. The alt.atheism category has a precision of 0.98, recall of 0.91 and f1-score

of 0.94 whilst the comp.graphics category has a lower precision of 0.93 but higher recall and f1-score of 0.99 and 0.96 respectively. Based on the confusion matrix in Appendix Figure 3.4.8, most misclassifications are alt.atheism posts being labelled incorrectly as comp.graphics posts with much fewer errors for the opposite.

The second pipeline introduces “cleanText” and this has led to a minor improvement in performance, as visualized in Appendix Figure 3.4.4. The results show that the accuracy increased to 95.19%, with the number of misclassifications of alt.atheism posts decreasing, as displayed in Appendix Figure 3.4.9. This indicates that the removal of email headers and addresses supports the model and allows it to focus on the message of each post, increasing accuracy. It is important to consider the wider picture – the misclassification of comp.graphics posts increased by 3. Therefore, with a decrease of 4 misclassified alt.atheism posts, this means the model does reinforce the fact that this cleaned text pipeline performs better overall.

The performance further enhances when lemmatization was incorporated alongside the “cleanText” method in the third pipeline. Looking at Appendix Figure 3.4.5, the accuracy rose to 95.62% - a more significant increased than the previous pipeline – with high precision and recall being maintained with both categories. Due to Appendix Figure 3.4.10, it is evident that misclassifications for both categories are reduced. This underlines how beneficial normalizing word forms to their lemmas is as this groups semantically similar terms due to the reduced vocabulary size.

Inspecting the fourth pipeline’s results, shown in Appendix Figure 3.4.6, which used the cleaned data with stemming rather than lemmatization, the accuracy increased to 96.04% (see Appendix Figure 3.4.11). Overall misclassification reduced by 3 which was not as expected – this proves that stemming is more effective than lemmatization in this dataset as the classification task was more successful with a higher accuracy rating.

The fifth and final pipeline combined lemmatization with stemming on the cleaned data and achieved the same exact accuracy and metrics as the fourth pipeline that only utilized stemming (see Appendix Figure 3.4.7). The misclassifications were also the same, as exemplified in Appendix Figure 3.4.12. This clearly concludes that, for this specific dataset and classification task, the integration of lemmatization to a pipeline that already uses stemming does not enhance error reduction or accuracy of the linear regression model.

Any additional normalization due to lemmatization is either not impactful enough or redundant based on the newsgroup post’s structure and vocabulary. This may also be due to the vocabulary and structure of the chosen posts where the more aggressive word reduction from stemming resulted in more consistent representation of features. It is crucial to note that stemming is also faster than lemmatization, stressing its efficiency compared to lemmatization in this case.

Overall, this spotlights that stemming and lemmatization, whilst offering valuable normalization techniques, do not always output improvements when combined. It is more crucial to examine the dataset and classification task as effectiveness of each method depends on the specific characteristics at hand. In this experiment, stemming extracts a sufficient amount of the morphological variation in the text data that the implementation of

lemmatization alone or alongside stemming does not further increase the model's capabilities in distinguishing between the two classes. See Table 3.4.1 for a summary of results.

Table 3.4.1: Evaluation Metrics Summary Table.

Pipeline	Accuracy	Precision	Recall	F1-Score
Baseline, processText only	95.05%	0.95	0.95	0.95
cleanText	95.19%	0.95	0.95	0.95
cleanText, Lemmatization	95.62%	0.96	0.96	0.96
cleanText, Stemming	96.04%	0.96	0.96	0.96
cleanText, Lemmatization, Stemming	96.04%	0.96	0.96	0.96

By conducting a qualitative analysis on the misclassified examples, it is revealed that the errors arise in posts that contain language or content that is ambiguous and/or overlap between the two classes. As an example, some alt.atheism posts include references to technical topics or terminology found typically in comp.graphics posts. This ends up confusing the classifier and to reinforce this conclusion one can look at the number of comp.graphics posts involving philosophical or religious content, explaining the faults in classification.

To summarize, these results confirm that thorough and careful preprocessing, generally the deduction of non-informative content and utilization of stemming, can considerably strengthen the text classification model's performance. As much as the baseline model performs very well already, the incremental enhancements from implementing further preprocessing indicates how valuable linguistic normalization is in the NLP pipeline. Another discovery of this experimentation is that stemming is preferred to lemmatization for this particular dataset, and that a combination of both techniques does not increase performance. This pipeline is validated to be effective for binary text classification tasks due to the evaluation yielding high accuracy, precision, recall and f1-score scores across both classes.

4. Conclusion

This report has exhibited the potency of combining systematic NLP preprocessing techniques along with machine learning of binary text classification. By applying a mix of linguistic analysis methods (tokenization, stop word removal, lemmatization, stemming, normalization), a plethora of insightful features were pulled from unstructured and hard to read newsgroup data. Through integrating these features into a TF-IDF weighted bag-of-words model and then classifying via logistic regression, high accuracy and balanced performance was achieved across both categories. Due to the comparative experimentation, it is ascertained that advanced preprocessing (mostly the removal of non-informative content and usage of stemming) delivered improvements over the baseline approach. It is crucial to experiment and thoroughly consider the specific dataset and task and tailor the preprocessing stage to these as combining stemming with lemmatization did not improve the performance of the classification model, as the expectation that lemmatization being more effective was proven to be false.

Even with the strong performance of the classification models, there are a few limitations – the experiment was limited to the two categories selected from the 20 Newsgroup dataset. This may not capture the whole diversity or complexity seen in larger or more varied text corpora. A non-linear classifier such as neural networks could provide improved performance, however as it is the current performance of the model is substantial and linear regression is much simpler, demanding less computational power. Overall, the findings emphasize the value of linguistic normalization and modular pipeline design in creating effective and interpretable text classification systems as well as establishing a base for subsequent exploration and experimentation in more complex models or extra feature engineering in the future.

References

Bird, S., Klein, E., & Loper, E. (2009). Natural Language Processing with Python. O'Reilly Media. Available at: https://www.nltk.org/book/ch06.html (Accessed 23 May)

Jurafsky, D. & Martin, J. H. (2023). Speech and Language Processing. (3rd ed.). Available at: https://web.stanford.edu/~jurafsky/slp3/ (Accessed 24 May)

Scikit-learn developers. (2024). Document classification with scikit-learn. Available at: https://scikitlearn.org/stable/auto_examples/text/plot_document_classification_20newsgroups.html (Accessed 23 May)

Appendix

Full skeleton code available here, followed by modified code in Figures.

```
from IPython import get_ipython
get_ipython().magic('reset -sf')
```

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
import pandas as pd
import numpy as np
categories = ['alt.atheism', 'soc.religion.christian', 'comp.graphics', 'sci.med']
from sklearn.datasets import fetch_20newsgroups
twenty_train = fetch_20newsgroups(subset='train',  categories=categories, shuffle=True, random_state=42)
twenty_test = fetch_20newsgroups(subset='test',  categories=categories, shuffle=True, random_state=42)
```

```
index=input('type your student number?')
```

```
type your student number?1234567
```

```
x=divmod(int(index),4)
yourdata1=x[1]
y=divmod(int(index),3)
yourdata2=y[1]

print('This is your data set index ----> (', x[1], y[1], ')' )
```

```
This is your data set index ----> ( 3 1 )
```

```
data1= twenty_train.target_names[x[1]]
data2= twenty_train.target_names[y[1]]
categories1=[data1,data2]
print(categories1)
```

```
# write your own NLP precessing examples with  preprocessing techniques.

dataset=twenty_train.data[1]
print(dataset)
# please   replace 1 in bracket to other data sample and explore the code

import nltk
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
```

```
# tokenize: search: nltk tokenize
example = "This is an example sentence."

from nltk.tokenize import word_tokenize
example_tokenize =word_tokenize(example)
#example_tokenize= word_tokenize(dataset) # replace example in bracket to dataset.
print("-------------------------tokenize:")
print(example_tokenize)
```

```
# stemmer: search: nltk stemmer
stemmer = nltk.stem.PorterStemmer()
example_stem = stemmer.stem(example)  # replace .....
print("-------------------------stem:")
print(example_stem)
```

```
# pos_taging: search: nltk pos tagging example
example_posTag=nltk.pos_tag(example_tokenize)
print("-------------------------pos_taging:")
print(example_posTag)
```

```
 # consituency parsing, chunking
grammar = "NP: {<DT>?<JJ>*<NN>}"
cp = nltk.RegexpParser(grammar)
result = cp.parse(example_posTag)
print(result)
```

```
import nltk
import re
from nltk.tokenize import sent_tokenize, word_tokenize
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
from tqdm import tqdm
from nltk.corpus import stopwords
stopwordEn = stopwords.words('english')
from nltk.corpus import wordnet
nltk.download('wordnet')
from nltk.stem.snowball import SnowballStemmer
stemmer = SnowballStemmer("english", ignore_stopwords=True)

def lemmaWord(word):
    lemma = wordnet.morphy(word)
    if lemma is not None:
        return lemma
    else:
        return word

def stemWord(word):
    stem = stemmer.stem(word)
    if stem is not None:
        return stem
    else:
        return word

def processText(text,lemma=False, gram=1, rmStop=True): # default remove stop words
    text = re.sub(r'(https|http)?:\/\/(\w|\.|\/|\?|\=|\&|\%)*\b|@\w+|#', '', text, flags=re.MULTILINE) #delete URL, #hashtag# , and @xxx
    tokens = word_tokenize(text)
    whitelist = ["n't", "not", "no"]
    new_tokens = []
    stoplist = stopwordEn if rmStop else []
    for i in tokens:
      i = i.lower()
      if i.isalpha() and (i not in stoplist or i in whitelist):  #i not in ['.',',',';']  and (...)
        if lemma: i = lemmaWord(i)
        new_tokens.append(i)
    del tokens
    # tokens = [lemmaWord(i.lower()) if lemma else i.lower() for i in tokens if (i.lower() not in stoplist or i.lower() in whitelist) and i.isalpha()]
    if gram<=1:
        return new_tokens
    else:
        return [' '.join(i) for i in nltk.ngrams(new_tokens, gram)]
```

```
def getTags(text):
  token = word_tokenize(text)
  token = [l.lower() for l in token]
  train_tags = nltk.pos_tag(token)
  return [i[1] for i in train_tags]
```

```
print(processText(dataset))
```

```
print(getTags(dataset))
```

```
print(categories1)
```

```python
#twenty_train1 = fetch_20newsgroups(subset='train',  categories=categories1, shuffle=True, random_state=42)
#twenty_test1 = fetch_20newsgroups(subset='test',  categories=categories1, shuffle=True, random_state=42)
```

```python
from sklearn.pipeline import Pipeline
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.linear_model import SGDClassifier, LogisticRegression

# Level: lexicon, model: tf-idf
text_clf = Pipeline([
    # add your code about text processing
    ('vect', CountVectorizer(analyzer=processText)),
    ('tfidf', TfidfTransformer(use_idf=True)),

    # change your classifier here, search: sklearn logistic regression example
    ('clf', SGDClassifier())
   #  ('clf', LogisticRegression())

])
```

```python
# To train the model
text_clf.fit(twenty_train.data, twenty_train.target)
```

```python
# To make prediction with dev/test set
predicted = text_clf.predict(twenty_test.data)
```

```python
# To evaluate your prediction on dev set
from sklearn import metrics
print("Accuracy:", metrics.accuracy_score(twenty_test.target, predicted))

print(metrics.classification_report(twenty_test.target, predicted, target_names=twenty_test.target_names))

# confusion class
pd.DataFrame(metrics.confusion_matrix(twenty_test.target, predicted),
             columns=twenty_test.target_names,index=twenty_test.target_names)
```

```python
df_pred = pd.DataFrame({'news':twenty_test.data,'prediction':predicted, 'true':twenty_test.target})
df_pred[df_pred['true'] != df_pred['prediction']]
```

```python
# Further preprocessing method to remove email headers and addresses
def cleanText(text):
    text = re.sub(r'^(From|Subject|Organization|Lines|Reply-To|NNTP-Posting-Host):.*$', '', text, flags=re.MULTILINE) # Remove headers (From:, Subject:)
    text = re.sub(r'\S+@\S+', '', text) # Remove full email addresses
    return text
```

Figure 3.2.1: “cleanText” Function.

```
# Show before/after cleaning text data

print("Original text:\n", fetch_20newsgroups(subset='train', categories=categories1).data[0][:500])
print("Cleaned text:\n", cleanText(fetch_20newsgroups(subset='train', categories=categories1).data[0])[:500])
```

```
Original text:
 From: frank@D012S658.uucp (Frank O'Dwyer)
Subject: Re: After 2000 years, can we say that Christian Morality is
Organization: Siemens-Nixdorf AG
Lines: 28
NNTP-Posting-Host: d012s658.ap.mchp.sni.de

In article <1993Apr15.125245.12872@abo.fi> MANDTBACKA@FINABO.ABO.FI (Mats Andtbacka) writes:
|In <1qie61$fkt@horus.ap.mchp.sni.de> frank@D012S658.uucp writes:
|> In article <30114@ursa.bear.com> halat@pooh.bears (Jim Halat) writes:
|
|> #I'm one of those people who does not know what the word objectiv
Cleaned text:




In article   (Mats Andtbacka) writes:
|In   writes:
|> In article   (Jim Halat) writes:
|
|> #I'm one of those people who does not know what the word objective means
|> #when put next to the word morality.  I assume its an idiom and cannot
|> #be defined by its separate terms.
|> #
|> #Give it a try.
|>
|> Objective morality is morality built from objective values.
|
|      "And these objective values are ... ?"
|Please be specific, and more importantly, motivate.

I'll take a wild guess
```

Figure 3.2.2: "cleanText" Function Removing Headers, Email Addresses and URLS.

```
# Base preprocessing method
def processText(text, lemma=False, stem=False, gram=1, rmStop=True): # default no lemma or stem, unigram and remove stop words
    text = re.sub(r'(https|http)?:\/\/(\w|\.|\/|\?|\=|\&|\%)*\b|@\w+|#', '', text, flags=re.MULTILINE)
    # The regex @\w+ matches an @ followed by word characters, not for full email addresses
    tokens = word_tokenize(text)
    whitelist = ["n't", "not", "no"]
    new_tokens = []
    stoplist = stopwordEn if rmStop else []
    for i in tokens:
        i = i.lower()
        if i.isalpha() and (i not in stoplist or i in whitelist):
            if lemma:
                i = lemmaWord(i)
            if stem:
                i = stemWord(i)
            new_tokens.append(i)
    del tokens
    if gram <= 1:
        return new_tokens
    else:
        return [' '.join(i) for i in nltk.ngrams(new_tokens, gram)]
```

Figure 3.2.3: "processText" Function Adopting Lemmatization, Removing Stop Words and Tokenizing Text.

```
sample_text = twenty_train1.data[0]

# Before processing
print("Before processing:\n", sample_text)

# After processing
print("After processing:\n", processText(sample_text))
```

```
Before processing:
 From: frank@D012S658.uucp (Frank O'Dwyer)
Subject: Re: After 2000 years, can we say that Christian Morality is
Organization: Siemens-Nixdorf AG
Lines: 28
NNTP-Posting-Host: d012s658.ap.mchp.sni.de

In article <1993Apr15.125245.12872@abo.fi> MANDTBACKA@FINABO.ABO.FI (Mats Andtbacka) writes:
|In <1qie61$fkt@horus.ap.mchp.sni.de> frank@D012S658.uucp writes:
|> In article <30114@ursa.bear.com> halat@pooh.bears (Jim Halat) writes:
|
|> #I'm one of those people who does not know what the word objective means
|> #when put next to the word morality.  I assume its an idiom and cannot
|> #be defined by its separate terms.
|> #
|> #Give it a try.
|>
|> Objective morality is morality built from objective values.
|
|      "And these objective values are ... ?"
|Please be specific, and more importantly, motivate.

I'll take a wild guess and say Freedom is objectively valuable.  I base
this on the assumption that if everyone in the world were deprived utterly
of their freedom (so that their every act was contrary to their volition),
almost all would want to complain.  Therefore I take it that to assert or
believe that "Freedom is not very valuable", when almost everyone can see
that it is, is every bit as absurd as to assert "it is not raining" on
a rainy day.  I take this to be a candidate for an objective value, and it
it is a necessary condition for objective morality that objective values
such as this exist.

--
Frank O'Dwyer                              'I'm not hatching That'
odwyer@sse.ie                              from "Hens",  by Evelyn Conlon

After processing:
 ['frank', 'subject', 'years', 'say', 'christian', 'morality', 'organization', 'ag', 'lines', 'article', 'mats', 'andtbacka', 'writes',
```

Figure 3.2.4: "processText" Example of a Cleaned Post Before and After Processing.

```
# Get POS tags method
def getTags(text):
  token = word_tokenize(text)
  token = [l.lower() for l in token]
  train_tags = nltk.pos_tag(token)
  return [i[1] for i in train_tags]
```

Figure 3.2.5: "getTags" Function

```
print("POS tags:\n", getTags(twenty_train1.data[0]))
```

```
POS tags:
 ['IN', ':', 'JJ', 'NN', 'NN', '(', 'JJ', 'NN', ')', 'NN', ':', 'NN', ':', 'IN', 'CD', 'NNS', ',', 'MD', 'PRP', 'VB', 'IN', 'JJ', 'NN', 'VBZ',
```

Figure 3.2.6: "getTags" Function Returning a Short List of Tokens with their POS Tags.

```
# Baseline - no cleaning, no lemma
baseline_pipeline = Pipeline([
    ('vect', CountVectorizer(analyzer=processText)),  # processText with default lemma=False
    ('tfidf', TfidfTransformer()), # TF-IDF Transformer
    ('clf', LogisticRegression(max_iter=1000)) # Logistic Regression classifier with max iterations set to 1000 to reach convergence
])


# With cleanText, no lemma
clean_pipeline = Pipeline([
    ('vect', CountVectorizer(analyzer=processText)),  # processText with default lemma=False
    ('tfidf', TfidfTransformer()),
    ('clf', LogisticRegression(max_iter=1000))
])


# With cleanText and lemma
lemma_pipeline = Pipeline([
    ('vect', CountVectorizer(analyzer=lambda x: processText(x, lemma=True))), # processText with lemma=True
    ('tfidf', TfidfTransformer()),
    ('clf', LogisticRegression(max_iter=1000))
])


# With cleanText and stem
stem_pipeline = Pipeline([
    ('vect', CountVectorizer(analyzer=lambda x: processText(x, stem=True))), # processText with stem=True
    ('tfidf', TfidfTransformer()),
    ('clf', LogisticRegression(max_iter=1000))
])


# With cleanText and lemma with stem
lemmastem_pipeline = Pipeline([
    ('vect', CountVectorizer(analyzer=lambda x: processText(x, lemma=True, stem=True))),# processText with lemma=True AND stem=True
    ('tfidf', TfidfTransformer()),
    ('clf', LogisticRegression(max_iter=1000))
])
```

Figure 3.3.1: Scikit-learn Pipeline Construction, Using "processText" for Every Document, TF-IDF Feature Vector Transformer and the Logistic Regression Classifier. Comments Detail Pipeline Differences.

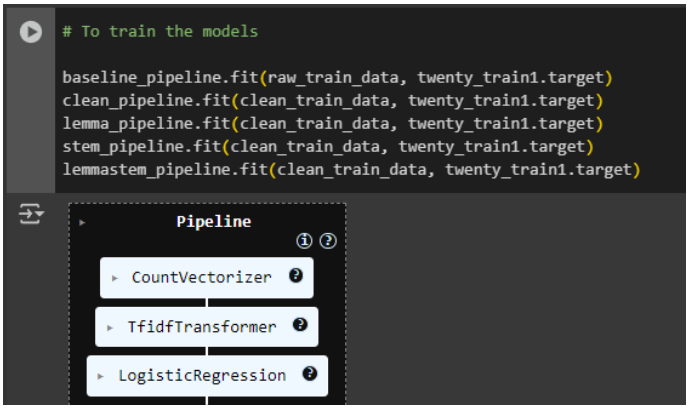

Figure 3.3.2: Model Training for Each Pipeline.

```python
# To make predictions with dev/test set

baseline_pred = baseline_pipeline.predict(raw_test_data)
clean_pred = clean_pipeline.predict(clean_test_data)
lemma_pred = lemma_pipeline.predict(clean_test_data)
stem_pred = stem_pipeline.predict(clean_test_data)
lemmastem_pred = lemmastem_pipeline.predict(clean_test_data)
```

Figure 3.3.3: Model Prediction for Each Pipeline.

```python
from sklearn.metrics import classification_report, accuracy_score, confusion_matrix # get evaluation metrics / error confusion matrix
import pandas as pd

# get all pipelines
pipelines = [
    ("Baseline", baseline_pred),
    ("With Clean Text", clean_pred),
    ("With Clean Text + Lemma", lemma_pred),
    ("With Clean Text + Stem", stem_pred),
    ("With Clean Text + Lemma + Stem", lemmastem_pred)
]

# for each pipeline, perform evaluation
for name, preds in pipelines:
    print(f"\n{name}:")
    print("Accuracy:", accuracy_score(twenty_test1.target, preds))
    print("Classification Report:")
    print(classification_report(twenty_test1.target, preds, target_names=twenty_test1.target_names))
```

Figure 3.4.1: Performance Evaluation for Each Pipeline's Model Predictions.

```python
# for each pipeline, get misclassified examples (5) and confusion matrix
for name, preds in pipelines:
    df_pred = pd.DataFrame({
        'news': twenty_test1.data,
        'prediction': preds,
        'true': twenty_test1.target
    })

    misclassified = df_pred[df_pred['true'] != df_pred['prediction']]
    print(f"\nMisclassified examples for {name}:")
    print(misclassified.head(5))  # Only first 5 to show all clearly

    print("Confusion Matrix:")
    print(pd.DataFrame(
        confusion_matrix(twenty_test1.target, preds),
        columns=twenty_test1.target_names,
        index=twenty_test1.target_names
    ))
```

Figure 3.4.2: Misclassification Identification and Confusion Matrix.

```
Baseline:
Accuracy: 0.9505649717514124
Classification Report:
              precision    recall  f1-score   support

  alt.atheism       0.98      0.91      0.94       319
comp.graphics       0.93      0.99      0.96       389

     accuracy                           0.95       708
    macro avg       0.96      0.95      0.95       708
 weighted avg       0.95      0.95      0.95       708
```

Figure 3.4.3: Performance Evaluation Results of Baseline Model.

```
With Clean Text:
Accuracy: 0.9519774011299436
Classification Report:
              precision    recall  f1-score   support

   alt.atheism       0.97      0.92      0.95       319
 comp.graphics       0.94      0.98      0.96       389

      accuracy                           0.95       708
     macro avg       0.95      0.95      0.95       708
  weighted avg       0.95      0.95      0.95       708
```

Figure 3.4.4: Performance Evaluation Results of Second Model.

```
With Clean Text + Lemma:
Accuracy: 0.9562146892655368
Classification Report:
              precision    recall  f1-score   support

   alt.atheism       0.98      0.92      0.95       319
 comp.graphics       0.94      0.98      0.96       389

      accuracy                           0.96       708
     macro avg       0.96      0.95      0.96       708
  weighted avg       0.96      0.96      0.96       708
```

Figure 3.4.5: Performance Evaluation Results of Third Model.

```
With Clean Text + Stem:
Accuracy: 0.96045197740113
Classification Report:
              precision    recall  f1-score   support

   alt.atheism       0.98      0.93      0.95       319
 comp.graphics       0.94      0.99      0.96       389

      accuracy                           0.96       708
     macro avg       0.96      0.96      0.96       708
  weighted avg       0.96      0.96      0.96       708
```

Figure 3.4.6: Performance Evaluation Results of Fourth Model.

```
With Clean Text + Lemma + Stem:
Accuracy: 0.96045197740113
Classification Report:
              precision    recall  f1-score   support

   alt.atheism       0.98      0.93      0.95       319
 comp.graphics       0.94      0.99      0.96       389

      accuracy                           0.96       708
     macro avg       0.96      0.96      0.96       708
  weighted avg       0.96      0.96      0.96       708
```

Figure 3.4.7: Performance Evaluation Results of Fifth Model.

```
Misclassified examples for Baseline:
                                                  news  prediction  true
0   From: aaron@minster.york.ac.uk\nSubject: Re: G...           1     0
4   Organization: Penn State University\nFrom: <SM...           1     0
15  From: jk87377@lehtori.cc.tut.fi (Kouhia Juhana...           0     1
20  From: kax@cs.nott.ac.uk (Kevin Anthoney)\nSubj...           1     0
82  From: aaron@minster.york.ac.uk\nSubject: Re: D...           1     0
Confusion Matrix:
               alt.atheism  comp.graphics
alt.atheism            289             30
comp.graphics            5            384
```

Figure 3.4.8: Misclassified Examples and Confusion Matrix for Baseline Model.

```
Misclassified examples for With Clean Text:
                                                  news  prediction  true
0   From: aaron@minster.york.ac.uk\nSubject: Re: G...           1     0
4   Organization: Penn State University\nFrom: <SM...           1     0
15  From: jk87377@lehtori.cc.tut.fi (Kouhia Juhana...           0     1
20  From: kax@cs.nott.ac.uk (Kevin Anthoney)\nSubj...           1     0
82  From: aaron@minster.york.ac.uk\nSubject: Re: D...           1     0
Confusion Matrix:
               alt.atheism  comp.graphics
alt.atheism            293             26
comp.graphics            8            381
```

Figure 3.4.8: Misclassified Examples and Confusion Matrix for Second Model.

```
Misclassified examples for With Clean Text + Lemma:
                                                  news  prediction  true
0   From: aaron@minster.york.ac.uk\nSubject: Re: G...           1     0
4   Organization: Penn State University\nFrom: <SM...           1     0
15  From: jk87377@lehtori.cc.tut.fi (Kouhia Juhana...           0     1
20  From: kax@cs.nott.ac.uk (Kevin Anthoney)\nSubj...           1     0
82  From: aaron@minster.york.ac.uk\nSubject: Re: D...           1     0
Confusion Matrix:
               alt.atheism  comp.graphics
alt.atheism            295             24
comp.graphics            7            382
```

Figure 3.4.9: Misclassified Examples and Confusion Matrix for Third Model.

```
Misclassified examples for With Clean Text + Stem:
                                                  news  prediction  true
0   From: aaron@minster.york.ac.uk\nSubject: Re: G...           1     0
4   Organization: Penn State University\nFrom: <SM...           1     0
15  From: jk87377@lehtori.cc.tut.fi (Kouhia Juhana...           0     1
20  From: kax@cs.nott.ac.uk (Kevin Anthoney)\nSubj...           1     0
82  From: aaron@minster.york.ac.uk\nSubject: Re: D...           1     0
Confusion Matrix:
               alt.atheism  comp.graphics
alt.atheism            296             23
comp.graphics            5            384
```

Figure 3.4.10: Misclassified Examples and Confusion Matrix for Fourth Model.

```
Misclassified examples for With Clean Text + Lemma + Stem:
                                                  news  prediction  true
0   From: aaron@minster.york.ac.uk\nSubject: Re: G...           1     0
4   Organization: Penn State University\nFrom: <SM...          1     0
15  From: jk87377@lehtori.cc.tut.fi (Kouhia Juhana...           0     1
20  From: kax@cs.nott.ac.uk (Kevin Anthoney)\nSubj...           1     0
82  From: aaron@minster.york.ac.uk\nSubject: Re: D...           1     0
Confusion Matrix:
               alt.atheism  comp.graphics
alt.atheism            296             23
comp.graphics            5            384
```

Figure 3.4.11: Misclassified Examples and Confusion Matrix for Fifth Model.