Twitter Sentiment Analysis using SVM

Python · Sentiment140 dataset with 1.6 million tweets

Date - 19 October 2023

Name :- Divyanshu Yadav

Enrollment No :- 10919011921

Branch :- Artificial Intelligence and Data Science

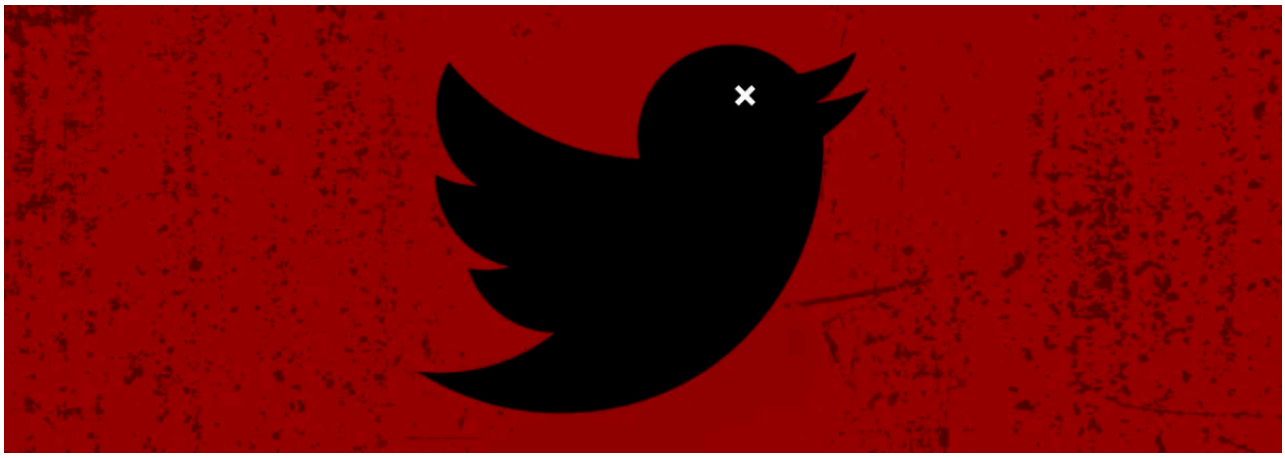
Batch :- B2-B

Kaggle Notebook Link :- https://www.kaggle.com/code/divzyzz/twitter-sentiment-analysis-using-svm-by-divyanshu/notebook

Abstract

Twitter sentiment analysis is a pivotal component in the realm of natural language processing and data analytics. It serves as a powerful tool for gauging the prevailing sentiment within the Twitterverse, a highly popular microblogging platform. This comprehensive report delves deep into the methodologies, use cases, challenges, and repercussions that underpin the practice of sentiment analysis on Twitter data.

Our exploration commences with a scrutiny of data acquisition and preprocessing techniques, underscoring the critical role of text cleaning in ensuring data quality. Subsequently, we delve into a spectrum of sentiment analysis methods, encompassing lexicon-based, machine learning-based, and hybrid approaches. The spectrum of applications for Twitter sentiment analysis is vast, with notable use cases spanning brand reputation management, political insights, customer support optimisation, and real-time event monitoring.

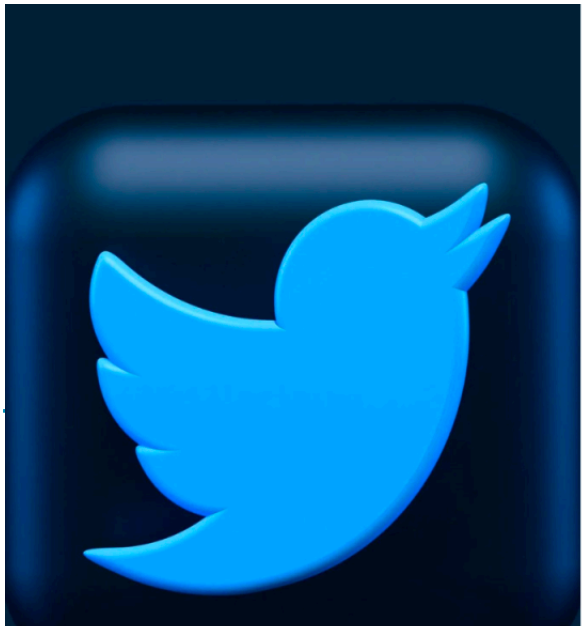


Nevertheless, there exists a series of challenges that necessitate attention, including the nuances of sarcasm interpretation, data quality assurance, bias mitigation, and navigating through linguistic ambiguity. Ethical considerations regarding user consent and data privacy are paramount, ensuring the responsible and lawful utilisation of data.

When executed skilfully, Twitter sentiment analysis yields invaluable insights that guide decision-making, safeguard brand reputation, and facilitate responsive engagement with customer feedback.

Introduction

Twitter is a microblogging site with a large user-generated material library on a variety of subjects, making it a great place to start when conducting sentiment analysis. A Natural Language Processing (NLP) technique called sentiment analysis, commonly referred to as opinion mining, tries to identify and quantify the sentiment expressed in text data, which can be either positive, negative, or neutral.



Methods

1. Data Acquisition:- Data is categorised into two types internal data and external data. Internal data that we already seen in the form of CSV files or within a database. In cases where we have limited data, we may employ data augmentation techniques to enhance it. On the other hand, external data entails the collection of information from external sources. This can involve activities such as web scraping from various websites or gathering real-time data through APIs(use huggingFace ,etc). Another approach to obtaining data is by leveraging publicly available datasets, often found on platforms like Kaggle. It's worth noting that when dealing with Twitter data, a common preprocessing step involves eliminating sources of noise, such as retweets, links, and non-textual elements, to ensure that the data is ready for analysis.

2.1 Text Cleaning

1. Removing Repeating character:-

```
#Cleaning and removing repeating characters
import re
def cleaning_repeating_char(text):
    return re.sub(r'(.)1+', r'1', text)
dataset['text'] = dataset['text'].apply(lambda x: cleaning_repeating_char(x))
dataset['text'].tail()
```

```
[18…  19995    Not much time off weekend work trip Malmï¿½ Fr...
      19996                                 One day holidays
      19997                    feeling right  hate DAMN HUMPREY
      19998    geezi hv READ whole book personality types emb...
      19999    I threw sign donnie bent over get but thingee ...
      Name: text, dtype: object
```

The code defines a function, `cleaning_repeating_char`, which uses regular expressions to replace consecutive repeating characters with a single occurrence. This function is then applied to the 'text' column of a dataset, effectively reducing repeated characters in the text data.

This cleaning operation helps to standardize and condense text, making it more amenable for subsequent text analysis or natural language processing tasks.

2. Removing URLS:-

```
def cleaning_URLs(data):
    return re.sub('((www.[^s]+)|(https?://[^s]+))',' ',data)
dataset['text'] = dataset['text'].apply(lambda x: cleaning_URLs(x))
dataset['text'].tail()
```

```
[19…  19995    Not much time off weekend work trip Malmï¿½ Fr...
      19996                                 One day holidays
      19997                    feeling right  hate DAMN HUMPREY
      19998    geezi hv READ whole book personality types emb...
      19999    I threw sign donnie bent over get but thingee ...
      Name: text, dtype: object
```

The code removes URLs from the 'text' column of the dataset using regular expressions, replacing them with spaces. This helps clean the text data for analysis by eliminating URLs.

2.2 Pre-processing

1. Removing Stop Words :-

```
stopwordlist = ['a', 'about', 'above', 'after', 'again', 'ain', 'all', 'am', 'an',
             'and','any','are', 'as', 'at', 'be', 'because', 'been', 'before',
             'being', 'below', 'between','both', 'by', 'can', 'd', 'did', 'do',
             'does', 'doing', 'down', 'during', 'each','few', 'for', 'from',
             'further', 'had', 'has', 'have', 'having', 'he', 'her', 'here',
             'hers', 'herself', 'him', 'himself', 'his', 'how', 'i', 'if', 'in',
             'into','is', 'it', 'its', 'itself', 'just', 'll', 'm', 'ma',
             'me', 'more', 'most','my', 'myself', 'now', 'o', 'of', 'on', 'once',
             'only', 'or', 'other', 'our', 'ours','ourselves', 'out', 'own', 're','s', 'same', 'she', "shes", 'should', "shouldve",'so', 'some',
             't', 'than', 'that', "thatll", 'the', 'their', 'theirs', 'them',
             'themselves', 'then', 'there', 'these', 'they', 'this', 'those',
             'through', 'to', 'too','under', 'until', 'up', 've', 'very', 'was',
             'we', 'were', 'what', 'when', 'where','which','while', 'who', 'whom',
             'why', 'will', 'with', 'won', 'y', 'you', "youd","youll", "youre",
             "youve", 'your', 'yours', 'yourself', 'yourselves']
```

+ Code + Markdown

```
[16]: #remove stopwords
STOPWORDS = set(stopwordlist)
def cleaning_stopwords(text):
    return " ".join([word for word in str(text).split() if word not in STOPWORDS])
dataset['text'] = dataset['text'].apply(lambda text: cleaning_stopwords(text))
dataset['text'].head()
```

```
[16… 800000              I LOVE @Health4UandPets u guys r best!!
800001    im meeting one besties tonight! Cant wait!! - ...
800002    @DaRealSunisaKim Thanks Twitter add, Sunisa! I...
800003    Being sick really cheap hurts much eat real fo...
800004                     @LovesBrooklyn2 effect everyone
Name: text, dtype: object
```

The code creates a list of common stopwords and defines a function, `cleaning_stopwords`, which removes these stopwords from text data. This function splits the text into words and only keeps words that are not in the list of stopwords. The 'text' column of the dataset is then processed using this function, resulting in text with stopwords removed.

2. Removing Punctuation:-

```
#code to clean and remove puncutation
import string
english_punctuations = string.punctuation
punctuations_list = english_punctuations
def cleaning_punctuations(text):
    translator = str.maketrans('', '', punctuations_list)
    return text.translate(translator)
dataset['text']= dataset['text'].apply(lambda x: cleaning_punctuations(x))
dataset['text'].tail()
```

```
[17… 19995    Not much time off weekend work trip Malmï¿½ Fr...
19996                                 One day holidays
19997                       feeling right  hate DAMN HUMPREY
19998    geezi hv READ whole book personality types emb...
19999    I threw sign donnie bent over get but thingee ...
Name: text, dtype: object
```

This code is designed to clean and remove punctuation from text data. It first defines a list of English punctuation symbols and then creates a function, `cleaning_punctuations`, that uses `str.maketrans()` and `translate()` to remove these punctuation symbols from the text. The 'text' column of the dataset is processed with this function, resulting in text with punctuation removed.

2.2 Pre-processing

3. Removing digits:-

```
#Cleaning and removing Numeric numbers
def cleaning_numbers(data):
    return re.sub('[0-9]+', '', data)
dataset['text'] = dataset['text'].apply(lambda x: cleaning_numbers(x))
dataset['text'].tail()
```

```
19995    Not much time off weekend work trip Malmï¿½ Fr...
19996                                  One day holidays
19997                    feeling right  hate DAMN HUMPREY
19998    geezi hv READ whole book personality types emb...
19999    I threw sign donnie bent over get but thingee ...
Name: text, dtype: object
```

This code is for cleaning and removing numeric numbers from text data. It defines a function, `cleaning_numbers`, which uses a regular expression to remove any sequences of digits (numeric numbers). The function is then applied to the 'text' column of the dataset, effectively removing numeric numbers from the text content.

4. Tokenization:-

Tokenization is the process of breaking down text into smaller units, often words or subwords, known as tokens in natural language processing (NLP). It is a fundamental step in text analysis and is used to facilitate various NLP tasks such as text processing, machine translation, sentiment analysis, and more. Tokens are the basic units that computers work with to understand natural language. Tokens can represent different types of linguistic units, depending on the level of granularity.

2.2 Pre-processing - Types of Tokenization

1. Word Tokenization: This is the most common type, where text is split into individual words. For example, the sentence "Tokenization is important" would be tokenized into three tokens: "Tokenization," "is," and "important."

2. Custom Tokenization: Custom tokenization techniques can be developed as needed. For instance, text can be tokenized based on particular delimiters or textual patterns.

3. Subword Tokenization: Subword tokenization splits text into smaller units that may not correspond to full words. This is useful in languages with complex morphology or for handling out-of-vocabulary words. Techniques like Byte-Pair Encoding (BPE) or Word Piece are commonly used for subword tokenization.

4. Sentence Tokenization: Sentence tokenization involves splitting a document into sentences. For instance, the paragraph "I love NLP. It's fascinating." is tokenized into two sentences: ["I love NLP.", "It's fascinating."]

5. Character Tokenization: In character tokenization, each character in the text becomes a token. For example, "Hello" would be tokenized as ["H", "e", "l", "l", "o"].

6. Tokenization Challenges: Tokenization can be challenging in languages with no clear word boundaries or in languages that use non-standard characters. It can also be complicated by punctuation, contractions, and abbreviations.

In summary, tokenization is a crucial step in NLP for computational analysis, as it converts text into a suitable format for further processing. The choice of tokenization method depends on the specific NLP task and the linguistic characteristics of the language being analysed.

```
#token get for next tweet
from nltk.tokenize import RegexpTokenizer
tokenizer = RegexpTokenizer('\s+', gaps = True)
dataset['text'] = dataset['text'].apply(tokenizer.tokenize)
```

```
dataset['text'].tail()
```

```
[22…  19995    [Not, much, time, off, weekend, work, trip, Ma...
      19996                                [One, day, holidays]
      19997                [feeling, right, hate, DAMN, HUMPREY]
      19998    [geezi, hv, READ, whole, book, personality, ty...
      19999    [I, threw, sign, donnie, bent, over, get, but,...
      Name: text, dtype: object
```

2.2 Pre-processing

5. Stemming :-

```
#stemming
import nltk
st = nltk.PorterStemmer()
def stemming_on_text(data):
    text = [st.stem(word) for word in data]
    return data
dataset['text']= dataset['text'].apply(lambda x: stemming_on_text(x))
dataset['text'].head()
```

```
[24…  800000          [I, LOVE, HealthUandPets, u, guys, r, best]
800001    [im, meeting, one, besties, tonight, Cant, wai...
800002    [DaRealSunisaKim, Thanks, Twitter, add, Sunisa...
800003    [Being, sick, really, cheap, hurts, much, eat,...
800004                     [LovesBrooklyn, effect, everyone]
Name: text, dtype: object
```

The code uses the Porter stemming algorithm from NLTK to reduce words to their root or base form. It applies stemming to the 'text' column of the dataset, making words more uniform for text analysis.

6. Lemmitization:-

```
n [24]:
lm = nltk.WordNetLemmatizer()
def lemmatizer_on_text(data):
    text = [lm.lemmatize(word) for word in data]
    return data
dataset['text'] = dataset['text'].apply(lambda x: lemmatizer_on_text(x))
dataset['text'].head()
```

```
)ut[24]:
800000            [love, healthuandpets, u, guys, r, best]
800001    [im, meeting, one, besties, tonight, cant, wai...
800002    [darealsunisakim, thanks, twitter, add, sunisa...
800003    [sick, really, cheap, hurts, much, eat, real, ...
800004                    [lovesbrooklyn, effect, everyone]
Name: text, dtype: object
```

The code employs the WordNet Lemmatizer from the NLTK library to perform lemmatization on the text data. It aims to reduce words to their base or dictionary form (lemma). The `lemmatizer_on_text` function is applied to the 'text' column of the dataset, lemmatizing individual words in the text, which can be advantageous for text analysis by converting words to their base form.

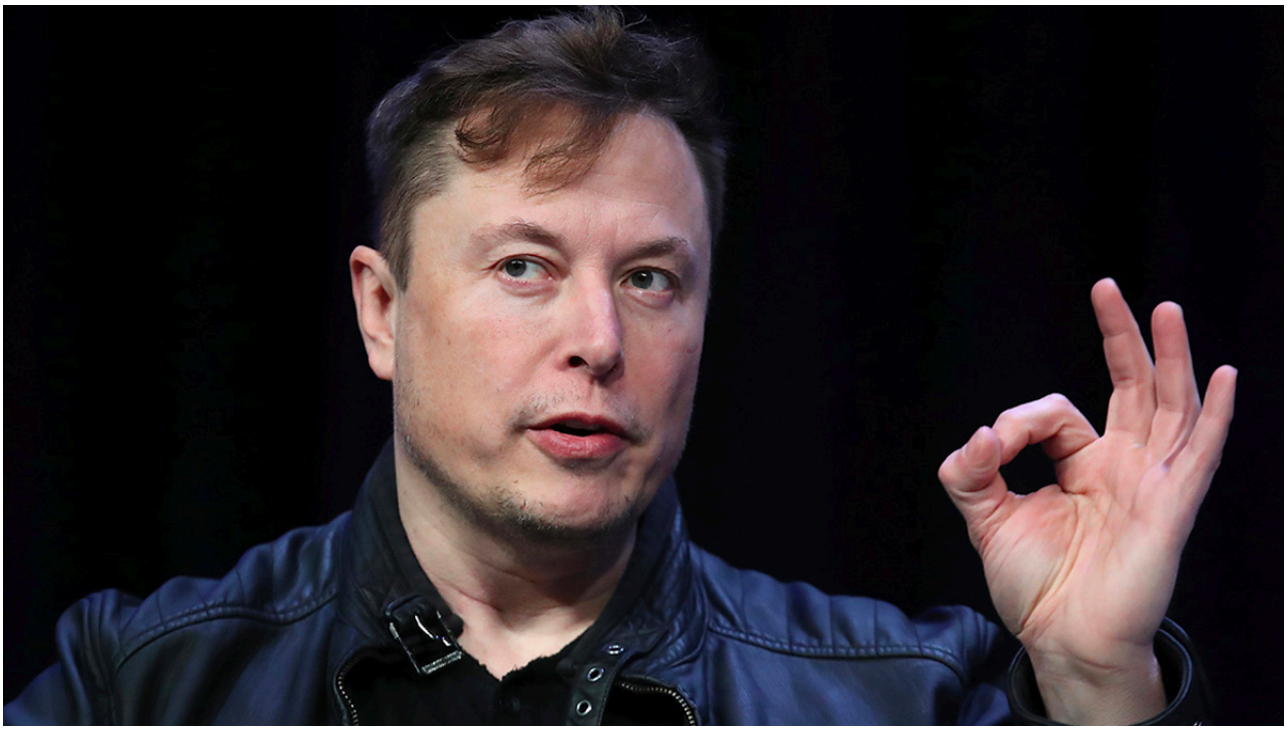
Feature Extraction

Feature extraction is the process of converting textual data into numerical representations, a crucial step in enabling machine learning algorithms to work with text. One widely used method for this conversion is TF-IDF, which stands for Term Frequency-Inverse Document Frequency. TF-IDF is a statistical measure employed in natural language processing and information retrieval to assess the significance of a term within a document concerning a broader collection of documents. It comprises two essential components: Term Frequency (TF) and Inverse Document Frequency (IDF).

Term Frequency (TF): TF quantifies how often a term appears in a document, measuring the frequency of a term 't' in document 'd' relative to the total number of terms in that document. A higher TF value signifies the importance of a term within that specific document.

Inverse Document Frequency (IDF): IDF evaluates the importance of a term across the entire corpus of documents by considering how unique or rare a term is. It reduces the weight of terms common across many documents while increasing the weight of rare, specific terms.

The TF-IDF score for a term in a document is the product of its TF and IDF values, indicating the term's relative importance within that specific document compared to its importance across the entire corpus. Terms with higher TF-IDF scores are considered more significant and relevant to the content of the document.



Use Cases

TF-IDF finds application in various NLP contexts, including:

1. Information Retrieval: TF-IDF aids search engines in ranking documents based on their relevance to a user's query.

2. Document Classification: It is used to categorise documents into predefined categories based on their content.

3. Text Summarisation: TF-IDF helps identify key sentences or phrases in a document for summarisation.

4. Keyword Extraction: It assists in extracting the most relevant keywords from a document.

5. Recommendation Systems: TF-IDF powers recommendation systems by suggesting documents or articles to users based on their preferences and content similarity.

6. Content Analysis: Researchers and analysts leverage TF-IDF to uncover key topics and themes within a collection of documents.

In summary, TF-IDF is an invaluable technique for quantifying term significance within documents concerning a broader corpus. It plays a fundamental role in various text analysis and retrieval tasks, making it an essential tool in the realm of natural language processing.

```
#Transforming Dataset using TF-IDF Vectorizer
from sklearn.feature_extraction.text import TfidfVectorizer
vectoriser = TfidfVectorizer(ngram_range=(1,2), max_features=5000)
vectoriser.fit(X_train)
print('No. of feature_words: ', len(vectoriser.get_feature_names_out()))
```

```
No. of feature_words:  5000
```

Modelling

```
from sklearn.svm import SVC
clf=SVC()
clf.fit(X_train,y_train)
y_pred=clf.predict(X_test)
```

There are several methods for performing sentiment analysis:

1. Lexicon-based Techniques: These methods assign sentiment scores to words by using sentiment lexicons (dictionaries of words with associated sentiment scores). The overall sentiment of a text is then computed by averaging these values.

2. Machine Learning-based Methods: To predict sentiment, these techniques rely on labeled data to train machine learning models such as Naive Bayes, Support Vector Machines (SVM), or more advanced deep learning models like Recurrent Neural Networks (RNNs) and Transformers.

3. Hybrid Approaches: Some approaches combine both lexicon-based and machine learning techniques to enhance accuracy.

In this report, we will utilise the Support Vector Machine (SVM), a machine learning algorithm, for sentiment analysis. SVM is a powerful method for classification tasks, including sentiment analysis, and it will be the focus of our analysis.

Evaluation

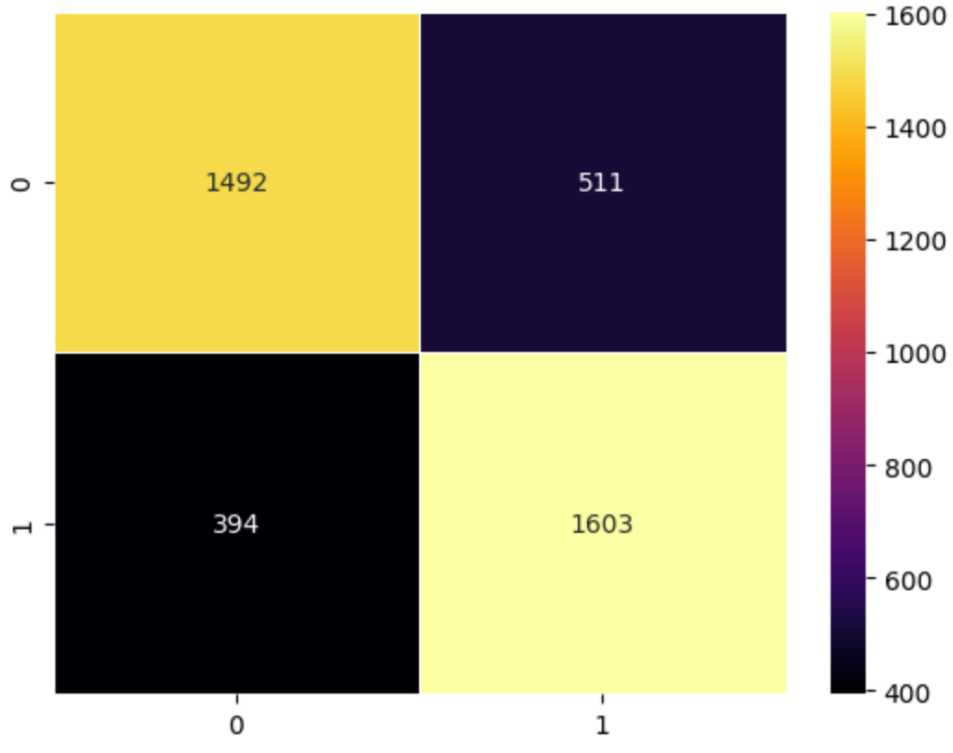
Confusion Matrix:-

```
from sklearn.metrics import accuracy_score, confusion_matrix

test_accuracy = accuracy_score(y_test, y_pred)
confusion = confusion_matrix(y_test, y_pred)

print("Test Accuracy:", test_accuracy)
print("Confusion Matrix:")
print(confusion)
```

```
Test Accuracy: 0.77375
Confusion Matrix:
[[1492  511]
 [ 394 1603]]
```

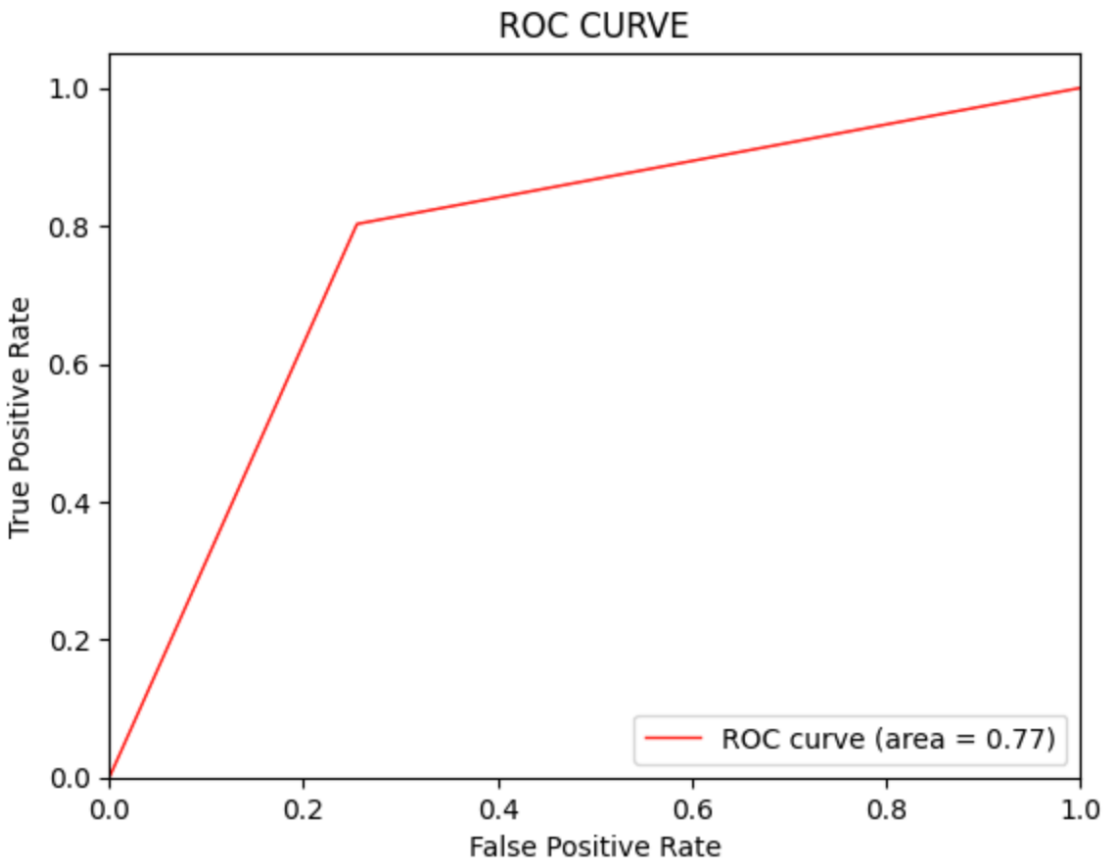
Heatmap:-

```
import seaborn as sns
custom_cmap = "magma"
sns.heatmap(cfm, annot=True, fmt='', linewidths=0.5, cmap=custom_cmap)
plt.show()
```



Accuracy

```
from sklearn.metrics import roc_curve, auc
fpr, tpr, thresholds = roc_curve(y_test, y_pred)
roc_auc = auc(fpr, tpr)
plt.figure()
plt.plot(fpr, tpr, color='red', lw=1, label='ROC curve (area = %0.2f)' % roc_auc)
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC CURVE')
plt.legend(loc="lower right")
plt.show()
```



Receiver Operating Characteristic (ROC) curve to assess the performance of the binary classification model. It calculates the false positive rate (FPR), true positive rate (TPR), and the area under the ROC curve (AUC). The AUC value is reported as 0.77, which indicates the model's ability to distinguish between the two classes.

Conclusion

In conclusion, sentiment analysis is a vital tool for understanding public sentiment, and it finds applications in various domains, from brand reputation management to politics and customer support. The process of sentiment analysis involves several key steps, including data collection, cleaning, and text processing.

Methods such as TF-IDF and machine learning algorithms are crucial for feature extraction and sentiment classification. SVM, in particular, is a robust choice for sentiment analysis tasks.

Moreover, tokenization, stemming, and lemmatization are essential text preprocessing techniques that contribute to the accuracy of sentiment analysis. These methods help convert text data into a suitable format for machine learning.

The ROC curve and AUC provide valuable insights into the model's classification performance, enabling further assessment of its effectiveness.

In summary, sentiment analysis serves as a valuable resource for decision-making, brand management, and understanding public sentiment in the age of social media and digital communication. The combination of techniques and methodologies outlined in this report equips analysts with the tools needed to extract meaningful insights from textual data and make informed decisions.