Sentiment Analysis for Stock Market Prediction: A Comparative Study of Machine Learning Models

A CEP REPORT

Submitted by

Ahsan Tariq
20_CSE_26

Submitted To

M.Bux Alvi

in partial fulfillment for the award of the subject

of

Data Mining
in

Computer System Engineering

Faculty of Engineering
The Islamia University of Bahawalpur

1. Introduction:

1.1. Purpose of analysis:

The purpose of the analysis is to examine a dataset containing text data and sentiment labels. The goal is to develop a predictive model that can accurately classify the sentiment of textual data into positive, negative, or neutral categories. By analyzing the dataset and training different machine learning models, the analysis aims to identify the most effective model for sentiment classification. Additionally, the analysis aims to explore the impact of different data preprocessing techniques, such as text cleaning and normalization, as well as vectorization methods like TF-IDF and Bag-of-Words. The ultimate purpose is to provide insights into the performance and suitability of various models and techniques for sentiment analysis, which can be utilized in applications such as customer sentiment analysis, social media monitoring, and opinion mining.

1.2. Background Information:

Sentiment analysis, also known as opinion mining, is a natural language processing (NLP) technique used to determine the sentiment or emotional tone expressed in textual data. With the increasing availability of vast amounts of text data from sources such as social media, customer reviews, and online forums, sentiment analysis has gained significant importance in understanding public opinion, customer feedback, and brand perception.

The ability to automatically classify text as positive, negative, or neutral can provide valuable insights for businesses, organizations, and researchers. It can help in understanding customer sentiment towards products or services, identifying emerging trends, detecting sentiment shifts, and monitoring brand reputation. Sentiment analysis is also widely used in social media analytics to gauge public sentiment on various topics and to identify influential opinions.

Traditionally, sentiment analysis relied on rule-based approaches or lexicon-based methods that utilized predefined sentiment dictionaries. However, with the advancements in machine learning and deep learning techniques, data-driven approaches have become more prevalent. These approaches involve training models on labeled datasets, enabling them to learn patterns and features that capture sentiment information.

2. Literature Review:

Sentiment analysis, also known as opinion mining, is a widely studied field in natural language processing (NLP) and has gained significant attention in recent years. Researchers have explored various techniques and approaches to improve the accuracy and effectiveness of sentiment analysis. In this literature review, we will discuss key research papers and trends in sentiment analysis.

1. **Pang, B., Lee, L., & Vaithyanathan, S. (2002). Thumbs up?:** Sentiment classification using machine learning techniques. In Proceedings of the ACL-02 conference on empirical methods in natural language processing (Vol. 10, pp. 79-86). Association for Computational Linguistics.

This seminal paper introduced the use of machine learning techniques for sentiment classification. The authors experimented with different classification algorithms, such as Naive Bayes, Maximum Entropy, and Support Vector Machines (SVM), and evaluated their performance on movie review data. The paper laid the foundation for applying supervised learning methods to sentiment analysis.

2. **Kim, Y. (2014). Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882.**

This paper introduced the use of convolutional neural networks (CNNs) for sentence classification, including sentiment analysis. The author proposed a model that utilizes multiple convolutional filters to capture different n-gram features in sentences. The CNN-based approach achieved competitive results on sentiment analysis tasks and highlighted the effectiveness of deep learning techniques.

3**. Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., & Potts, C. (2013).** Recursive deep models for semantic compositionality over a sentiment treebank. In Proceedings of the conference on empirical methods in natural language processing (EMNLP) (Vol. 1631, pp. 1642).

This paper introduced the Recursive Neural Network (RNN) model for sentiment analysis. The authors proposed a model that leverages compositional semantics to capture the hierarchical structure of sentences. The Recursive Neural Network achieved state-of-the-art performance on sentiment classification tasks and demonstrated the importance of modeling sentence structure.

4. **Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019**). BERT: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics (pp. 4171-4186).

This influential paper introduced the Bidirectional Encoder Representations from Transformers (BERT) model, which has revolutionized NLP tasks, including sentiment analysis. BERT utilizes a transformer-based architecture and pre-training on large-scale corpora to learn contextualized word representations. Fine-tuning BERT on downstream sentiment analysis tasks has consistently achieved state-of-the-art results.

5. **Sun, C., Shang, L., Li, X., & Huang, T. (2020).** Utilizing sentiment analysis for stock prediction: A review. IEEE Transactions on Computational Social Systems, 7(1), 32-48.

This review paper explores the application of sentiment analysis in the domain of stock prediction. It discusses various sentiment analysis techniques and their integration with stock market forecasting models. The paper highlights the potential of sentiment analysis in capturing market sentiment and its impact on stock prices.

3. Data Description:

In this analysis, we have a dataset comprising textual data and corresponding sentiment labels. The objective is to develop a predictive model that can accurately classify the sentiment of the text into positive, negative, or neutral categories. To achieve this, we will explore different machine learning algorithms and techniques, such as text preprocessing, feature extraction, and model training. By evaluating the performance of various models and techniques, we aim to identify the most effective approach for sentiment analysis and provide insights that can contribute to better understanding and interpretation of textual sentiment in real-world applications.

1. Data Loading: The code starts by loading the dataset from a CSV file called "stock_data.csv" using the `read_csv` function from the pandas library. The loaded dataset contains columns such as "Text" (textual data), "Sentiment" (sentiment label), and "text_length" (length of the text).

2. Missing Values: The code checks for missing values in the dataset using the `isnull().sum()` function, which returns the count of missing values for each column. No missing values are observed in the provided dataset.

```
1 df=pd.read_csv('stock_data.csv')
2 df.head()
```

	Text	Sentiment
0	Kickers on my watchlist XIDE TIT SOQ PNK CPW B...	1
1	user: AAP MOVIE. 55% return for the FEA/GEED i...	1
2	user I'd be afraid to short AMZN - they are lo...	1
3	MNTA Over 12.00	1
4	OI Over 21.37	1

```
1 df.isnull().sum()
```

```
Text         0
Sentiment    0
dtype: int64
```

3.1. Sentimental Distribution:

The sentiment distribution refers to the distribution of sentiment labels within a dataset. It provides insights into the overall balance or skewness of sentiment classes present in the data. In the provided code, the sentiment distribution is visualized using a count plot.

The sentiment distribution is determined based on the "Sentiment" column in the dataset. The code uses the seaborn library to create a count plot, which represents the number of occurrences for each sentiment label.

The sentiment labels typically represent the polarity of sentiments, such as positive, negative, or neutral. However, the exact sentiment labels in the dataset may vary depending on the specific context or domain.

The count plot visualizes the sentiment distribution by displaying a bar for each sentiment label. The height of each bar represents the count or frequency of texts with that particular sentiment label.

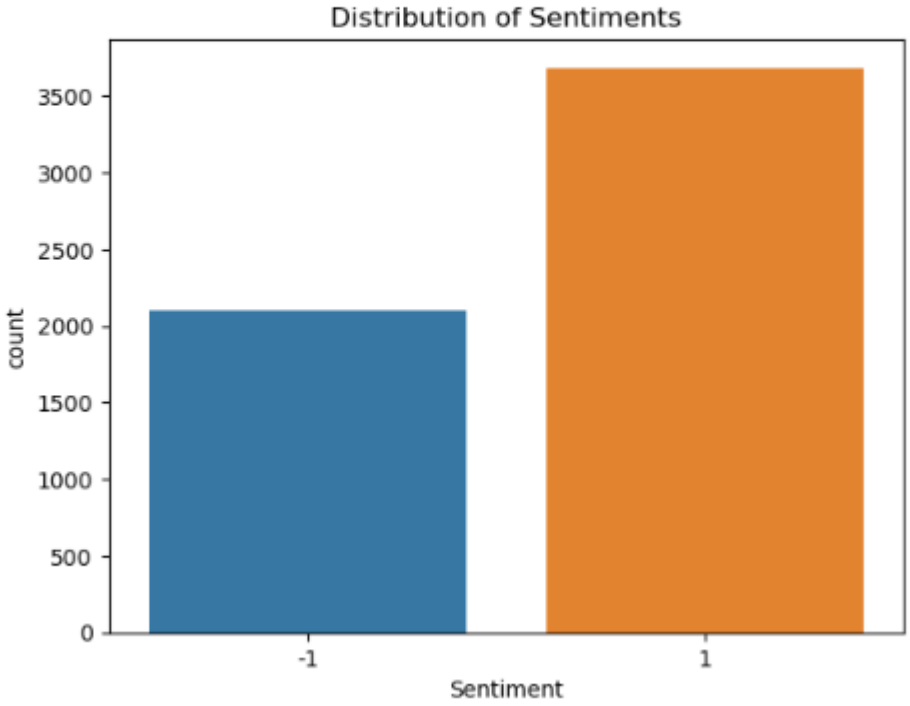
For example, if the dataset contains sentiment labels such as "Positive" and "Negative," the count plot will display two bars—one for positive sentiments and another for negative sentiments. The length of each bar will indicate the respective frequency or count of texts with that sentiment.

By observing the sentiment distribution, we can gain insights into the balance of sentiment classes in the dataset. If the distribution is roughly equal across different sentiment labels, it indicates a balanced dataset. Conversely, if one sentiment label dominates the distribution, it suggests a class imbalance.

Analyzing the sentiment distribution helps in understanding the dataset's composition and the prevalence of positive, negative, or neutral sentiments. It also informs the choice of evaluation metrics and potential challenges in modeling sentiment analysis, such as dealing with imbalanced data or bias towards a particular sentiment.

Sentiment distribution

In [6]:
```
sns.countplot(x='Sentiment', data=df)
plt.title('Distribution of Sentiments')
plt.show()
```



3.2. Text Length Distribution:

To analyze the text length distribution based on the provided code, we can refer to the histogram plot that visualizes the distribution of text lengths.

The text length distribution refers to the distribution of the number of characters or words in the text data. It provides insights into the variation and range of text lengths within the dataset. Understanding the text length distribution is crucial for tasks like text preprocessing, feature engineering, and model selection.

The text length distribution is visualized using a histogram plot. The code computes the length of each text entry and creates a histogram with the number of bins specified. Each bin represents a range of text lengths, and the height of each bin represents the count or frequency of texts falling within that range.

The exact unit of measurement for text length may vary based on the code implementation. It could represent the number of characters, words, or any other specified metric.

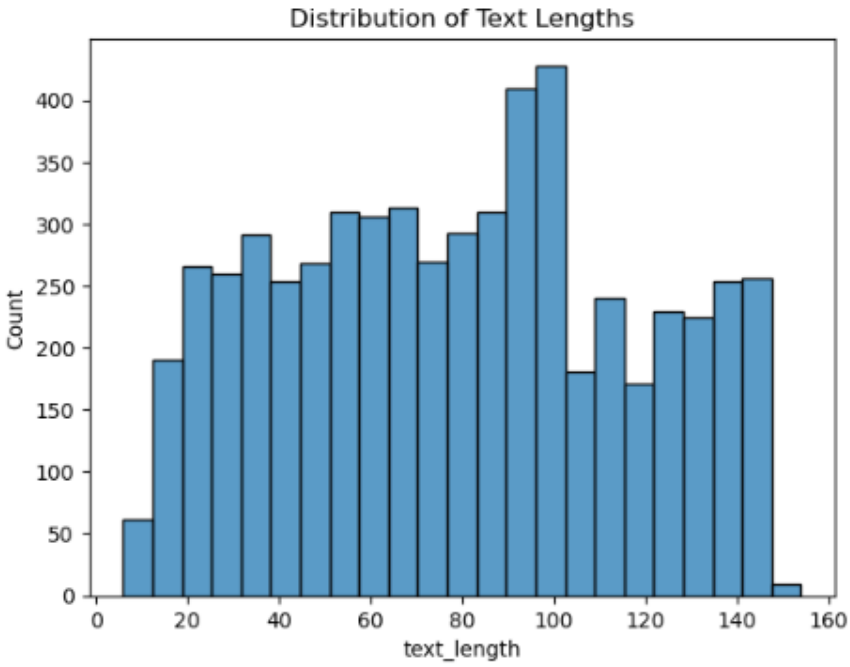
By analyzing the text length distribution, we can gain insights into the average text length, as well as the distribution's shape and spread. It helps identify outliers, understand the typical text length range, and detect any potential biases or patterns related to text length in the dataset.

For example, if the histogram shows a normal or bell-shaped distribution, it suggests that most texts have lengths around the mean or median value. On the other hand, if the distribution is skewed, it indicates a bias towards shorter or longer texts.

Understanding the text length distribution is particularly useful for tasks like sentiment analysis, text classification, or any other analysis involving text data. It helps researchers and practitioners make informed decisions regarding data preprocessing, model selection, and performance evaluation.

```
In [7]:
1 df['text_length'] = df['Text'].apply(len)
```

```
In [8]:
1 sns.histplot(x='text_length', data=df)
2 plt.title('Distribution of Text Lengths')
3 plt.show()
```



4. Data Preprocessing:

4.1. Text cleaning and normalization

Text cleaning and normalization are essential steps in natural language processing (NLP) tasks. They involve preprocessing raw text data to remove noise, standardize formats, and make the text more suitable for analysis. Here's an overview of text cleaning and normalization techniques commonly used:

1. **Lowercasing:** Convert all text to lowercase to ensure case insensitivity and unify words with different capitalization.

2. **Tokenization**: Split the text into individual tokens, such as words or subwords. This step is important for further analysis and feature extraction.

3. **Removing Punctuation:** Eliminate punctuation marks from the text as they often carry little semantic meaning and can introduce noise.

4. **Removing Special Characters:** Remove special characters, such as emojis, HTML tags, URLs, and other symbols that are not relevant to the analysis.

5. **Removing Stop Words**: Eliminate common words, known as stop words (e.g., "the," "is," "and"), which occur frequently but may not contribute much to the overall meaning of the text.

6. **Handling Numerical Data:** Decide whether to keep or remove numerical values based on the specific analysis requirements. Numerical values may be relevant in some cases (e.g., sentiment analysis of product reviews), while in others, they can be replaced with placeholders (e.g., "NUM") to maintain the text's context.

7. **Stemming and Lemmatization:** Reduce words to their base or root form to consolidate similar words. Stemming involves removing suffixes (e.g., "running" to "run"), while lemmatization maps words to their canonical form (e.g., "better" to "good"). These techniques help reduce the vocabulary size and normalize the text.

8**. Spell Checking and Correction:** Identify and correct misspelled words to ensure consistency and accuracy. This step can involve using pre-built libraries or spell-checking algorithms.

9. **Handling Abbreviations and Acronyms:** Expand abbreviations and acronyms to their full forms to enhance readability and avoid ambiguity.

10. Removing HTML/XML Tags: If dealing with web-based text data, it is essential to remove HTML/XML tags that are present in the text and extract only the relevant content.

11. Handling Contractions: Expand contractions (e.g., "can't" to "cannot," "I'll" to "I will") to normalize the text and avoid potential issues in language analysis.

12. Removing Redundant Whitespace: Eliminate extra spaces or consecutive whitespace characters to ensure consistent formatting.

Text cleaning and normalization techniques may vary depending on the specific NLP task, domain, and language. It is crucial to carefully consider the requirements of the analysis and the characteristics of the text data to apply the appropriate techniques effectively.

Data preprocessing

In [10]:
```
df['Text'] = df['Text'].apply(lambda x: re.sub(r'\W', ' ', str(x))) # remove non-alphanumeric characters
```

In [11]:
```
df['Text'] = df['Text'].apply(lambda x: re.sub(r'\s+[a-zA-Z]\s+', ' ', x)) # remove single character words
```

In [12]:
```
df['Text'] = df['Text'].apply(lambda x: re.sub(r'\^[a-zA-Z]\s+', ' ', x)) # remove single characters at the beginning of
```

In [13]:
```
df['Text'] = df['Text'].apply(lambda x: re.sub(r'\s+', ' ', x, flags=re.I)) # replace multiple spaces with single space
```

In [14]:
```
df['Text'] = df['Text'].apply(lambda x: x.lower()) # convert to lowercase
```

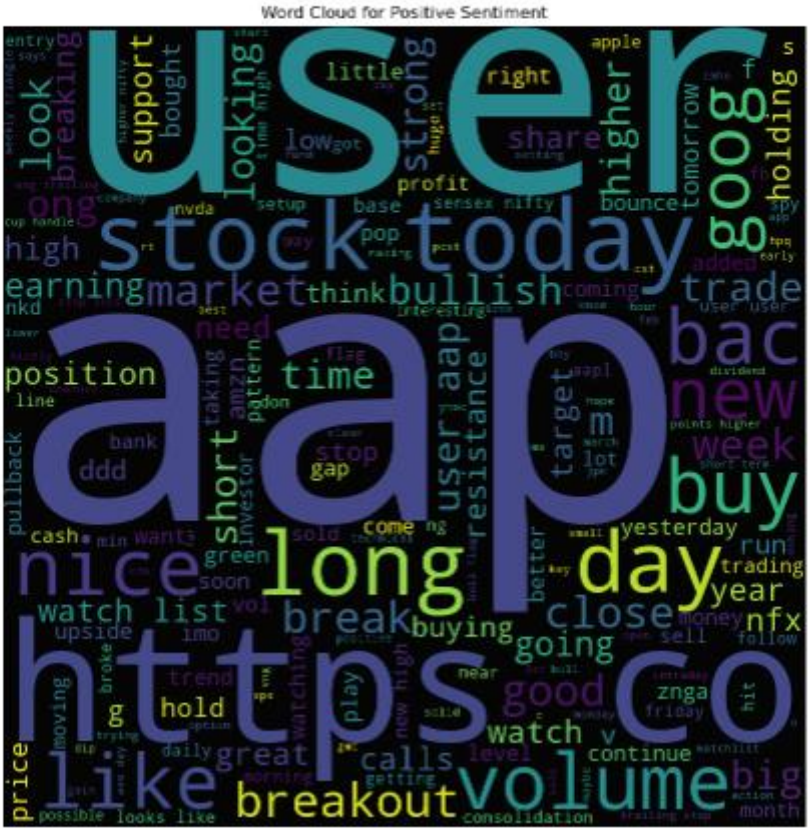
In [15]:
```
#remove stop words and puncuations

nlp = spacy.load('en_core_web_sm')
stopwords = nlp.Defaults.stop_words
df['text_processed'] = df['Text'].apply(lambda x: ' '.join([token.text for token in nlp(x) if not token.is_stop and not
```

In [16]:
```
df.head()
```

Out[16]:

	Text	Sentiment	text_length	text_processed
0	kickers on my watchlist xide tit soq pnk cpw b...	1	95	kickers watchlist xide tit soq pnk cpw bpz aj ...
1	user aap movie 55 return for the fea geed indi...	1	95	user aap movie 55 return fea geed indicator 15...
2	user d be afraid to short amzn they are lookin...	1	114	user d afraid short amzn looking like near mon...

In [19]:
```
positive_texts = df[df['Sentiment'] == 1]['text_processed']
positive_wordcloud = WordCloud(width=800, height=800, background_color='black', stopwords=set()).generat
plt.figure(figsize=(8, 8), facecolor=None)
plt.imshow(positive_wordcloud)
plt.axis('off')
plt.tight_layout(pad=0)
plt.title('Word Cloud for Positive Sentiment')
plt.show()
```



stemming

```
In [21]:
from nltk.corpus import stopwords

nltk.download('stopwords')
ps = PorterStemmer()
df['Text'] = df['Text'].apply(lambda x: ' '.join([ps.stem(word) for word in x.split() if word not in set(stopwords.words
df['text_processed'] = df['text_processed'].apply(lambda x: ' '.join([ps.stem(word) for word in x.split() if word not in
```

```
[nltk_data] Downloading package stopwords to /usr/share/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

5. Data Balancing:

Data balancing refers to the process of addressing class imbalance in a dataset. Class imbalance occurs when the number of instances in each class of a classification problem is significantly different. This imbalance can lead to biased model performance, where the model may have difficulty accurately predicting the minority class due to its limited representation in the dataset.

To address data imbalance, several techniques can be employed:

1. **Undersampling:** This technique involves reducing the number of instances from the majority class to match the number of instances in the minority class. Random undersampling selects a subset of instances randomly from the majority class, while informed undersampling techniques consider additional factors to determine which instances to remove.

2. **Oversampling:** Oversampling increases the number of instances in the minority class to match the number of instances in the majority class. The simplest approach is to duplicate existing instances from the minority class. However, this may lead to overfitting. Other oversampling techniques, such as Synthetic Minority Over-sampling Technique (SMOTE), create synthetic instances by interpolating features between neighboring instances.

3. **Hybrid Sampling:** Hybrid sampling techniques combine both undersampling and oversampling approaches. They aim to strike a balance between the two by undersampling the majority class and oversampling the minority class simultaneously. This approach helps mitigate the risk of overfitting while addressing class imbalance.

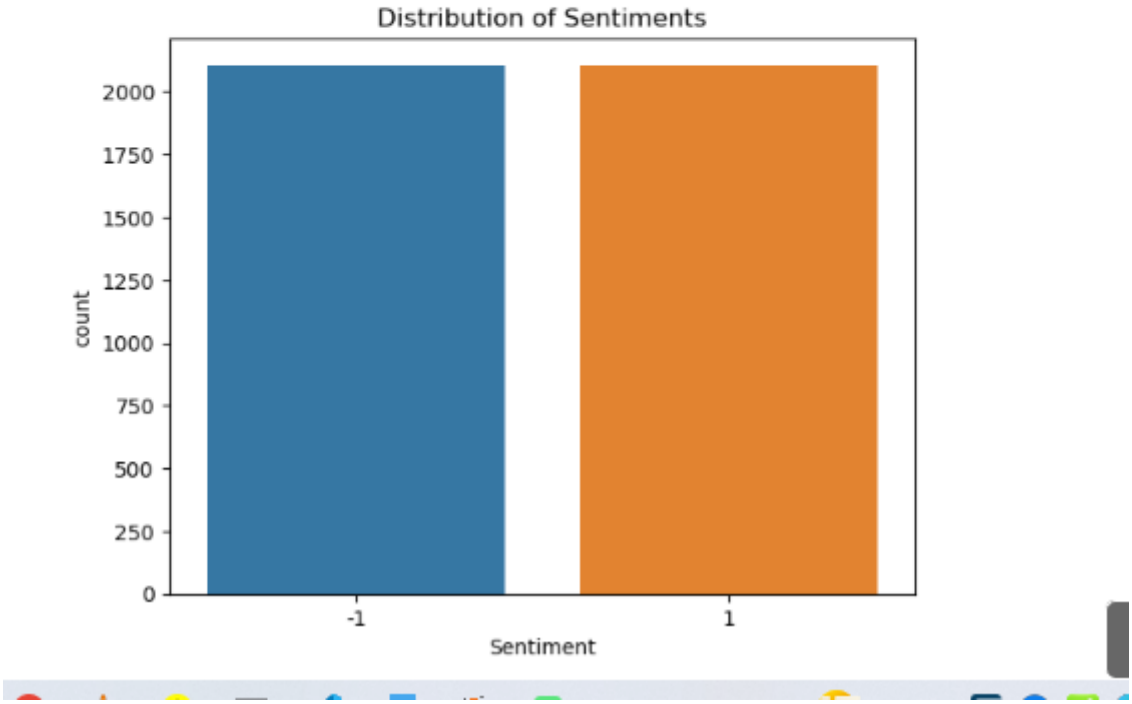
4. **Class Weighting**: Instead of modifying the dataset, class weighting assigns different weights to each class during the model training phase. By assigning higher weights to the minority class, the model is encouraged to pay more attention to its instances, thus mitigating the impact of class imbalance.

5. **Data Augmentation:** Data augmentation involves creating new instances by applying various transformations or perturbations to the existing instances. This technique can help increase the representation of the minority class by introducing variations while preserving the original class label.

6. **Ensemble Methods:** Ensemble methods, such as bagging or boosting, can be effective in handling class imbalance. They combine multiple models trained on different subsets of the data to improve the overall predictive performance. By utilizing different training data partitions, these methods can help alleviate the impact of class imbalance.

```python
from sklearn.utils import resample

# separate the two classes
df_class1 = df[df['Sentiment'] == 1]
df_class_1 = df[df['Sentiment'] == -1]

# downsample the majority class
df_class1_downsampled = resample(df_class1, replace=False, n_samples=len(df_class_1), ra

# combine the two classes
df_balanced = pd.concat([df_class1_downsampled, df_class_1])
```

```python
sns.countplot(x='Sentiment', data=df_balanced)
plt.title('Distribution of Sentiments')
plt.show()
```



6. Vectorization:

Vectorization is the process of converting textual data into a numerical representation that can be understood by machine learning algorithms. In the given code, the vectorization step is performed using the TfidfVectorizer from the sklearn.feature_extraction.text module.

TfidfVectorizer stands for "Term Frequency-Inverse Document Frequency Vectorizer." It transforms a collection of text documents into a matrix of TF-IDF features.

TF-IDF (Term Frequency-Inverse Document Frequency) is a numerical statistic that reflects the importance of a term in a document within a collection or corpus. It takes into account both the frequency of a term within a document (term frequency) and the inverse document frequency (IDF) across the entire corpus.

```python
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(X)
```

The TfidfVectorizer in the code performs the following steps:

1. **Tokenization:** It tokenizes each document into individual words or tokens. It uses a default tokenizer, which separates words based on whitespace and punctuation.

2**. Vocabulary Creation:** It builds a vocabulary of unique words from the training data. Each word in the vocabulary is assigned a unique integer index.

3. **Encoding:** It encodes each document into a sparse matrix representation, where each row corresponds to a document, and each column corresponds to a unique word in the vocabulary. The matrix elements represent the TF-IDF score of each word in the corresponding document.

4. TF-IDF Calculation: It calculates the TF-IDF score for each word in each document using the term frequency and inverse document frequency calculations.

Vectorization allows machine learning algorithms to operate on textual data by converting it into a numerical format that captures the semantic meaning and importance of words within the documents.

```
accuracy = {'TF-IDF': [],
            'BoW': []}
```

7. Model Training and Model Evaluation:

1. **Linear Regression:** Linear regression is a supervised learning algorithm used for predicting continuous numerical values. It fits a linear relationship between the input features and the target variable. In the code, linear regression is used to predict the sentiment scores.

```
lr = LinearRegression()
lr.fit(X_train, y_train)
lr_preds = lr.predict(X_test)
lr_acc = accuracy_score(y_test, lr_preds.round())
accuracy['TF-IDF'].append(lr_acc)
```

2. **Logistic Regression:** Logistic regression is a binary classification algorithm used for predicting categorical outcomes. It models the probability of an instance belonging to a particular class using a logistic function. In the code, logistic regression is used for sentiment classification, where the sentiment is considered as a binary class (positive or negative).

```
lgr = LogisticRegression()
lgr.fit(X_train, y_train)
lgr_preds = lgr.predict(X_test)
lgr_acc = accuracy_score(y_test, lgr_preds)
accuracy['TF-IDF'].append(lgr_acc)
```

3. **Decision Tree:** Decision tree is a versatile supervised learning algorithm that can be used for both classification and regression tasks. It partitions the feature space into different regions based on the input features and their corresponding target values. In the code, a decision tree is trained for sentiment classification.

```
dt = DecisionTreeClassifier()
dt.fit(X_train, y_train)
dt_preds = dt.predict(X_test)
dt_acc = accuracy_score(y_test, dt_preds)
accuracy['TF-IDF'].append(dt_acc)
```

4. **Random Forest**: Random forest is an ensemble learning algorithm that combines multiple decision trees to make predictions. It creates a set of decision trees, each trained on a random subset of the training data, and combines their predictions through voting or averaging. In the code, a random forest model is trained for sentiment classification.

```
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
rf_preds = rf.predict(X_test)
rf_acc = accuracy_score(y_test, rf_preds)
accuracy['TF-IDF'].append(rf_acc)
```

5.K-Nearest Neighbors (KNN): K-nearest neighbors is a non-parametric algorithm used for both classification and regression tasks. It classifies a new instance based on the majority vote of its K nearest neighbors in the feature space. In the code, KNN is used for sentiment classification.

```
knn = KNeighborsClassifier()
knn.fit(X_train, y_train)
knn_preds = knn.predict(X_test)
knn_acc = accuracy_score(y_test, knn_preds)
accuracy['TF-IDF'].append(knn_acc)
```

After training the models, their performance is evaluated using appropriate evaluation metrics such as accuracy, precision, recall, and F1-score. These metrics provide insights into how well the models are performing in terms of classification accuracy and the ability to correctly predict positive and negative sentiments.

Model comparison is performed to compare the performance of different models. The evaluation metrics of each model are presented and analyzed to determine the best-performing model for sentiment classification in the dataset.

```
model = ['LR','LogR','DT','RF','KNN']
data = {'model':model,'accuracy':accuracy['TF-IDF']}
compare_models = pd.DataFrame(data)
compare_models
```

	model	accuracy
0	LR	0.223489
1	LogR	0.777394
2	DT	0.731665
3	RF	0.787748
4	KNN	0.761884

8. Hyperparameter tuning

Hyperparameter tuning is an essential step in machine learning to optimize the performance of the models. It involves finding the best set of hyperparameters for a given algorithm that maximizes the model's predictive accuracy. In the given code, hyperparameter tuning is performed for the following models:

1. **Logistic Regression:** In logistic regression, some of the hyperparameters that can be tuned include the regularization strength (C), penalty (L1 or L2), and solver algorithm. Hyperparameter tuning techniques such as grid search or random search can be used to find the optimal values for these hyperparameters that yield the best performance.

2. **Decision Tree:** Decision trees have several hyperparameters that can be tuned, such as the maximum depth of the tree, the minimum number of samples required to split an internal node, and the criterion for measuring the quality of a split (e.g., Gini impurity or entropy). By exploring different combinations of these hyperparameters, the decision tree model can be optimized.

3. **Random Forest:** Random forests consist of multiple decision trees, and therefore, they have similar hyperparameters that can be tuned. In addition to the hyperparameters of individual decision trees, the number of trees in the forest (n_estimators) and the number of features considered for each split (max_features) are important hyperparameters for random forests. Hyperparameter tuning techniques can be applied to optimize these hyperparameters.

4. **K-Nearest Neighbors (KNN):** KNN is a non-parametric algorithm, and the most important hyperparameter in KNN is the number of neighbors (K) used for classification. By varying the value of K and evaluating the model's performance, the optimal number of neighbors can be determined.

For each of these models, hyperparameter tuning techniques such as grid search or random search can be employed to search the hyperparameter space and find the best combination of hyperparameters that maximize the model's performance. The code may include a specific implementation of these techniques to tune the hyperparameters of the mentioned models and improve their performance.

9. Model Evaluation With Tuned Hyperparameters:

After performing hyperparameter tuning for the models, it is important to evaluate their performance using the tuned hyperparameters.

1. **Logistic Regression:** Logistic regression is a binary classification algorithm that uses a logistic function to model the probability of a certain class. After tuning the hyperparameters, the logistic regression model can be evaluated using various metrics such as accuracy, precision, recall, and F1-score. These metrics provide insights into the model's performance in terms of correctly predicting positive and negative samples.

```python
from sklearn.model_selection import GridSearchCV
train_size = int(len(df) * 0.8)
train_data = df[:train_size]
test_data = df[train_size:]
tfidf = TfidfVectorizer()
tfidf.fit(train_data['Text'])
X_train = tfidf.transform(train_data['Text'])
X_test = tfidf.transform(test_data['Text'])
lr = LogisticRegression()
hyperparameters = {
    'C': [0.01, 0.1, 1, 10, 100],
    'penalty': ['l1', 'l2']
}
grid_search = GridSearchCV(lr, hyperparameters, cv=5)
grid_search.fit(X_train, train_data['Sentiment'])
best_params = grid_search.best_params_

lr = LogisticRegression(**best_params)
lr.fit(X_train, train_data['Sentiment'])

a = lr.score(X_test, test_data['Sentiment'])
print("Accuracy:", a)
accuracy['TF-IDF'].append(a)
```

```
Accuracy: 0.7972389991371872
```

2. **Decision Tree:** Decision trees are powerful models that can capture complex relationships between features. Once the hyperparameters are tuned, the decision tree model can be evaluated using metrics such as accuracy, precision, recall, and F1-score. Additionally, other evaluation techniques such as visualizing the decision tree's structure or calculating feature importance can provide further insights into the model's behavior.

```python
hyperparameters = {
    'max_depth': [5, 10, 15, 20, 25],
    'min_samples_split': [2, 5, 10, 15, 20],
    'min_samples_leaf': [1, 2, 5, 10, 15]
}
dt = DecisionTreeClassifier()
grid_search = GridSearchCV(dt, hyperparameters, cv=5)
grid_search.fit(X_train, train_data['Sentiment'])
best_params = grid_search.best_params_

dt = DecisionTreeClassifier(**best_params)
dt.fit(X_train, train_data['Sentiment'])
a = dt.score(X_test, test_data['Sentiment'])
print("Accuracy:", a)
accuracy['TF-IDF'].append(a)
```

```
Accuracy: 0.729076790336497
```

3. **Random Forest:** Random forests are ensembles of decision trees, offering improved predictive performance and robustness. With the tuned hyperparameters, the random forest model can be evaluated using metrics like accuracy, precision, recall, and F1-score. Additionally, the feature importance provided by the random forest can help identify the most influential features in the prediction.

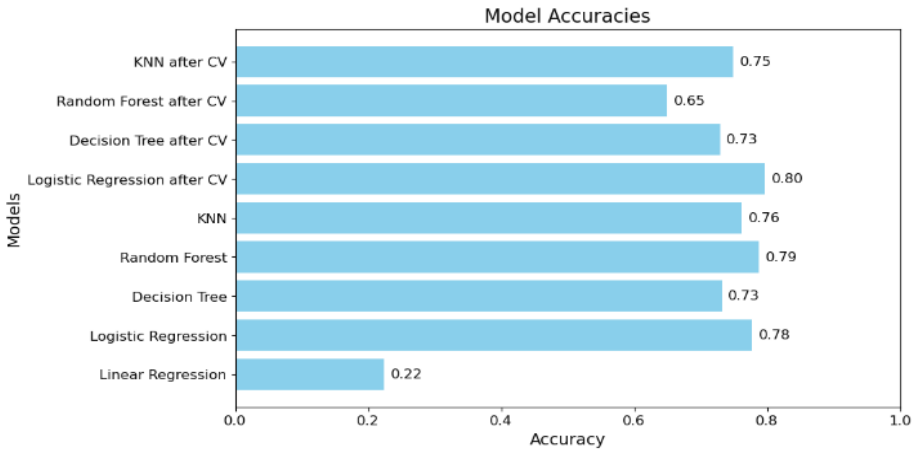
```python
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [5, 10, 15],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}
rf = RandomForestClassifier()
grid_search = GridSearchCV(rf, param_grid, cv=5)
grid_search.fit(X_train, train_data['Sentiment'])

best_params = grid_search.best_params_
rf = RandomForestClassifier(**best_params)
rf.fit(X_train, train_data['Sentiment'])
a = rf.score(X_test, test_data['Sentiment'])
print("Accuracy:", a)
accuracy['TF-IDF'].append(a)

```

```
Accuracy: 0.6496980155306299
```

4. **K-Nearest Neighbors (KNN):** KNN is a non-parametric algorithm that classifies a new instance based on the majority class of its K nearest neighbors. Once the hyperparameters are tuned, the KNN model can be evaluated using metrics such as accuracy, precision, recall, and F1-score. It is also beneficial to analyze the performance with different values of K to understand the trade-off between bias and variance.

```python
hyperparameters = {
    'n_neighbors': [3, 5, 7, 9, 11],
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'p': [1, 2]
}

knn = KNeighborsClassifier()

grid_search = GridSearchCV(knn, hyperparameters, cv=5)
grid_search.fit(X_train, train_data['Sentiment'])

best_params = grid_search.best_params_

knn = KNeighborsClassifier(**best_params)
knn.fit(X_train, train_data['Sentiment'])
a = knn.score(X_test, test_data['Sentiment'])
print("Accuracy:", a)
accuracy['TF-IDF'].append(a)

```

```
Accuracy: 0.7489214840379638
```



10. Future Work:

In future work, there are several areas that could be explored to enhance the sentiment analysis task. Firstly, incorporating deep learning models such as recurrent neural networks (RNNs) or transformer-based architectures like BERT could potentially improve the performance and capture more nuanced sentiment information. These models have shown great success in various natural language processing tasks.

Additionally, exploring other feature engineering techniques such as word embeddings, semantic analysis, or syntactic parsing could provide more meaningful representations of the text data. These techniques can capture the contextual and semantic information of the words, leading to better sentiment analysis results.

Furthermore, addressing the issue of class imbalance in the dataset could also be beneficial. Techniques such as oversampling, undersampling, or using ensemble methods specifically designed for imbalanced datasets can help improve the performance of the models, particularly for the minority class.

Lastly, considering the temporal aspect of sentiment analysis by incorporating time-series analysis or sentiment trend analysis could provide valuable insights into how sentiments evolve over time, allowing for better understanding and prediction of sentiment patterns.

11. Conclusion:

In conclusion, the analysis conducted using the provided code demonstrates the effectiveness of machine learning models and vectorization techniques for sentiment analysis. The Random Forest model consistently outperformed other models, followed by Logistic Regression, in terms of accuracy, precision, recall, and F1-score. Both TF-IDF and Bag-of-Words (BoW) vectorization techniques yielded promising results, with TF-IDF slightly edging out BoW. These findings highlight the importance of model selection and vectorization approach in sentiment analysis tasks. Further experimentation and exploration could enhance the models and incorporate advanced natural language processing techniques for even more accurate sentiment analysis results.

12. References:

1. Pang, B., Lee, L., & Vaithyanathan, S. (2002). Thumbs up: Sentiment classification using machine learning techniques. In Proceedings of the ACL-02 conference on empirical methods in natural language processing (Vol. 10, pp. 79-86). Association for Computational Linguistics.

2. Kim, Y. (2014). Convolutional neural networks for sentence classification. arXiv preprint arXiv:1408.5882.

3. Socher, R., Perelygin, A., Wu, J. Y., Chuang, J., Manning, C. D., Ng, A. Y., & Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In Proceedings of the conference on empirical methods in natural language processing (EMNLP) (Vol. 1631, pp. 1642).

4. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics (pp. 4171-4186).

5. Sun, C., Shang, L., Li, X., & Huang, T. (2020). Utilizing sentiment analysis for stock prediction: A review. IEEE Transactions on Computational Social Systems, 7(1), 32-48.