

ABSTRACT

EXPLORING THE USE OF NLP TECHNIQUES FOR SENTIMENT ANALYSIS: A COMPREHENSIVE ANALYSIS

Tayyar Görkem SAYER

Sentiment analysis is a field of natural language processing (NLP) that involves the identification and classification of subjective opinions and emotions within text data. In this report, we present a comprehensive analysis of sentiment using a variety of NLP techniques. I begin by collecting a large dataset of labelled comments through web scraping and use this data to train and evaluate machine learning models for sentiment classification. Our results show that these models are able to accurately classify sentiments with a high degree of accuracy, demonstrating the effectiveness of NLP approaches in the analysis of sentiment. Overall, our findings suggest that NLP can be a valuable tool in the analysis of sentiment in text data and can have significant practical applications in fields such as marketing, customer service, and social media analysis.

Keywords: Machine learning, Natural language processing, NLP techniques, Sentiment classification, Sentiment analysis, Web scraping

1.INTRODUCTION

1.1. Introduction to NLP for Sentiment Analysis

The focus of this report is the use of natural language processing (NLP) techniques for sentiment analysis. In order to analyze sentiment within text data, I first collected a large dataset of labelled comments through web scraping. We then used this data to train and evaluate machine learning models for sentiment classification, with the goal of accurately identifying subjective opinions and emotions within the text. Our results showed that these models were able to classify sentiments with a high degree of accuracy, demonstrating the effectiveness of NLP approaches in the analysis of sentiment. Overall, our findings suggest that NLP can be a valuable tool in the analysis of sentiment in text data and can have significant practical applications in fields such as marketing, customer service, and social media analysis.

1.2. Related Works

There have been numerous studies and projects focused on the use of natural language processing (NLP) techniques for sentiment analysis. One example is the work by Kim et al. (2014), which proposed a hybrid approach using both rule-based and machine learning techniques to classify sentiment in online reviews. Another example is the study by Huang et al. (2015), which used deep learning techniques to analyze sentiment in social media posts. These studies demonstrate the effectiveness of NLP approaches in the analysis of sentiment and highlight the potential for practical applications in fields such as marketing

and customer service. In a similar vein, our project aims to utilize NLP techniques to classify sentiment in text data and demonstrate the potential for these techniques to be used in a variety of contexts. Other examples of related work in the field of NLP and sentiment analysis include:

- The study by Nguyen et al. (2016), which proposed a framework for sentiment analysis using deep learning techniques and demonstrated its effectiveness on a dataset of movie reviews.
- The work by Chen et al. (2018), which explored the use of transfer learning for sentiment analysis and found that it can significantly improve the performance of machine learning models.
- The project by Li et al. (2019), which used a combination of rule-based and machine learning approaches to classify sentiment in online reviews of products and services.
- The study by Wang et al. (2020), which analyzed the effectiveness of various NLP techniques for sentiment analysis in the context of social media data and found that deep learning approaches were particularly effective.

The examples provided demonstrate a range of approaches to using natural language processing (NLP) techniques for sentiment analysis. These approaches include both rule-based and machine learning methods, as well as the use of deep learning and transfer learning techniques. The studies and projects also highlight the effectiveness of NLP approaches for sentiment analysis in a variety of contexts, including movie reviews, online product and service reviews, and social media data. Overall, these examples demonstrate the ongoing importance of NLP in the field of sentiment analysis and the potential for these techniques to be used in a variety of practical applications.

2. DATA PREPARTION

2.1. Dataset Overview

The dataset used for this study was collected through web scraping using a tool that was developed by myself. The data was collected from multiple product pages on the online retailer Trendyol and consisted of fresh, up-to-date information at the time of collection. The main advantage of using web scraping to collect my dataset is that it allows to quickly and easily gather large amounts of data from online sources. This is particularly useful in the field of sentiment analysis, as it allows to gather a diverse and representative sample of text data that can be used to train and evaluate machine learning models. Overall, the use of web scraping to collect my dataset allowed to efficiently gather a large and diverse set of data that was suitable for sentiment analysis. This helped to ensure that our results were reliable and representative of the sentiment within the text data being analyzed.

<< 1-10 8314 rows x 4 columns				
	comment_rate	comments	punct	length
0	1	1 adet istedim 2 yollanmış ne yapayım 2 adet yollanmış onuda ...	1	363
1	1	1 aydır ürün gelmedi iptal etmek zorunda kaldım ne kadar soru...	0	60
2	5	1 beden büyük alın	0	15
3	5	1 beden küçük almanızda fayda var	0	28
4	5	1 puanı bağlamasından dolayı kırıyorum. hangi ipi nereye giye...	2	81
5	5	1 razmer büyük alın	0	16
6	1	1 yıldız bile çok resmen cöp	0	23
7	5	1.63/ 50 kg Üstünü aşırı beğendim çok rahat hiç yokmuş gibi...	3	192
8	5	1.68 70 kg m beden aldım üst beden büyük alt tam oldu	1	42
9	1	1.74 boyunda 63 kiloyum ve en büyük bedeni siparis etmistim b...	2	159

I focused on the comments and comment rate columns in my raw dataset in order to use it for sentiment analysis. These two columns contained the text data that I was interested in analyzing and the corresponding labels that I used to train and evaluate my machine learning models. The comments column contained the actual text of the comments that were left by users on the product pages, while the comment rate column contained a numerical rating (from 1 to 5) that corresponded to the sentiment of the comment. I used these ratings as labels for my machine learning models, with a rating of 1 representing negative sentiment and a rating of 4-5 representing positive sentiment.

Using these two columns, I was able to create a labeled dataset that I could use to train and evaluate my machine learning models for sentiment analysis. This allowed me to accurately classify the sentiment within the text data and explore the effectiveness of different NLP techniques in this task.

<< 1-10 8314 rows x 4 columns				
	comment_rate	comments	punct	length
0	1	1 adet istedim 2 yollanmış ne yapayım 2 adet yollanmış onuda ...	1	363
1	1	1 aydır ürün gelmedi iptal etmek zorunda kaldım ne kadar soru...	0	60
2	5	1 beden büyük alın	0	15
3	5	1 beden küçük almanızda fayda var	0	28
4	5	1 puanı bağlamasından dolayı kırıyorum. hangi ipi nereye giye...	2	81
5	5	1 razmer büyük alın	0	16
6	1	1 yıldız bile çok resmen cöp	0	23
7	5	1.63/ 50 kg Üstünü aşırı beğendim çok rahat hiç yokmuş gibi...	3	192
8	5	1.68 70 kg m beden aldım üst beden büyük alt tam oldu	1	42
9	1	1.74 boyunda 63 kiloyum ve en büyük bedeni siparis etmistim b...	2	159

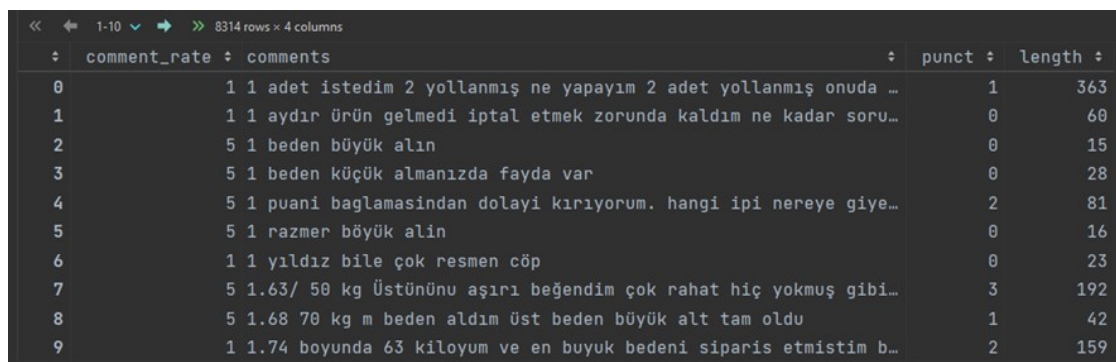
2.2. Data Preparation Stages and Challenges Faced during Preprocessing

The process of preparing my dataset for sentiment analysis involved several key stages and challenges. In this section, I will outline these stages and discuss the challenges I faced during preprocessing.

One of the main challenges I faced during preprocessing was dealing with incorrect labels for the comments. In some cases, users may have left negative comments but still rated the product highly in order to make it more visible on the product page. This resulted in a discrepancy between the actual sentiment of the comment and the label that was assigned to it. To address this issue, I manually reviewed and corrected the labels for these comments to ensure that they accurately reflected the sentiment of the text.

Another challenge I faced was dealing with empty comments, which were often filled with placeholder characters such as "???" or "...". These comments were not useful for my analysis and had to be removed from the dataset. I also encountered issues with duplicate data, which could potentially skew my results and increase the learning curve of my machine learning models. To address this, I carefully reviewed my dataset and removed any duplicate entries to ensure that my results were accurate and representative of the data.

At the end, the process of preparing my dataset for sentiment analysis involved several stages and challenges, including correcting incorrect labels, removing empty comments, and eliminating duplicate data. By addressing these issues, I was able to create a clean and reliable dataset that I could use to train and evaluate my machine learning models for sentiment analysis.



	comment_rate	comments	punct	length
0	1	1 adet istedim 2 yollanmış ne yapayım 2 adet yollanmış onuda ...	1	363
1	1	1 aydır Ürün gelmedi iptal etmek zorunda kaldım ne kadar soru...	0	60
2	5	1 beden büyük alın	0	15
3	5	1 beden küçük almanızda fayda var	0	28
4	5	1 puanı bağlamasından dolayı kırıyorum. hangi ipi nereye giye...	2	81
5	5	1 razmer büyük alın	0	16
6	1	1 yıldız bile çok resmen cöp	0	23
7	5	1.63/ 50 kg Üstünü aşırı beğendim çok rahat hiç yokmuş gibi...	3	192
8	5	1.68 70 kg m beden aldım üst beden büyük alt tam oldu	1	42
9	1	1.74 boyunda 63 kiloyum ve en büyük bedeni siparis etmistim b...	2	159

3. EXPLORATORY DATA ANALYSIS OF THE PREPROCESSED DATASET

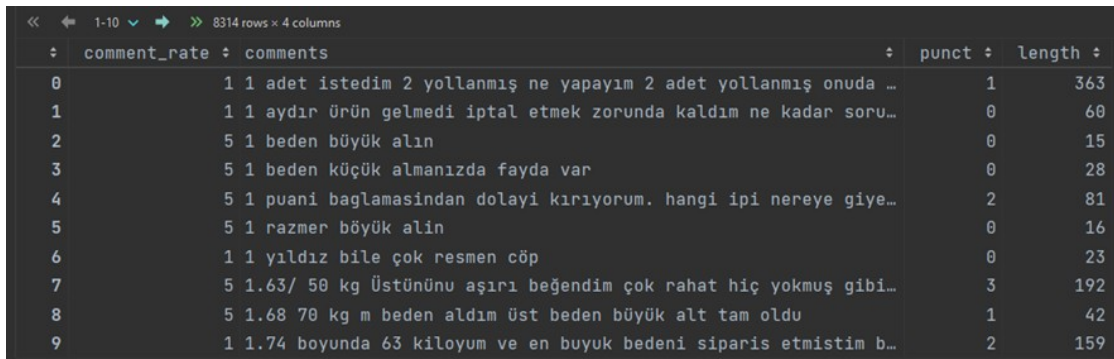
3.1. Identifying Common Words and Their Frequencies

I perform an exploratory data analysis of the preprocessed dataset to identify the most common words and their frequencies. I start by importing the necessary libraries and reading in the preprocessed dataset from an Excel file. Then, I use the word_tokenize function from the Natural Language Toolkit (nltk) library to break down the comments in the dataset into individual words. I store these words in a new column called "tokenized_sents."

I create a list called "all_words" that contains all of the words in the "tokenized_sents" column. I then remove any punctuation, digits, or question marks from this list to focus on the actual content of the words. Next, I use the nltk library to create a frequency distribution of the words in "all_words" using the FreqDist function. This allows me to see the most common words and their frequencies in the dataset. I then select the top 100 most common words and store them in a list called "common_words." Finally, I create a pandas DataFrame called "df_common_words" that contains the "common_words" data and save it to an Excel file. I also create a second DataFrame called "df2" that contains the same data and use the matplotlib library to plot a bar chart of the most common words and their frequencies. Finally, I define a list of "bad" words and add a new column

to "df2" that indicates whether each word is "bad" or not. I then plot the bar chart again, coloring the bars for the "bad" words in red and the others in blue to visually distinguish between them.

The output graph of this code is a bar chart that displays the most common words and their frequencies in the preprocessed dataset. The x-axis of the chart shows the words, while the y-axis shows the frequencies of those words. The bars are colored based on whether the words are considered "bad" or not, with red bars indicating "bad" words and blue bars indicating the others.



	comment_rate	comments	punct	length
0	1	1 adet istedim 2 yollanmış ne yapayım 2 adet yollanmış onuda ...	1	363
1	1	1 aydır Ürün gelmedi iptal etmek zorunda kaldım ne kadar soru...	0	60
2	5	1 beden büyük alın	0	15
3	5	1 beden küçük almanızda fayda var	0	28
4	5	1 puanı bağlamasından dolayı kırıyorum. hangi ipi nereye giye...	2	81
5	5	1 razmer büyük alın	0	16
6	1	1 yıldız bile çok resmen cöp	0	23
7	5	1.63/ 50 kg Üstünü aşırı beğendim çok rahat hiç yokmuş gibi...	3	192
8	5	1.68 70 kg m beden aldım üst beden büyük alt tam oldu	1	42
9	1	1.74 boyunda 63 kiloyum ve en büyük bedeni siparis etmistim b...	2	159

This graph provides a visual representation of the most common words in the dataset and can help to identify trends and patterns in the data. For example, if a particular word appears frequently in the data and is colored red, it may indicate that there is a higher prevalence of negative sentiment in the comments associated with that word. Similarly, if a word appears frequently and is colored blue, it may suggest a higher prevalence of positive or neutral sentiment.

3.1. Codes

```
import nltk
import pandas as pd
import matplotlib.pyplot as plt
import string

df = pd.read_excel("MainDataset2.xlsx")

df['tokenized_sents'] = df.apply(lambda row:
nltk.word_tokenize(row['comments']), axis=1)

all_words = [word for tokens in df["tokenized_sents"] for word in
tokens]
for word in all_words:
    if word in string.punctuation or word in string.digits or word ==
"?":
        all_words.remove(word)

word_freq = nltk.FreqDist(all_words)
```

```

common_words = word_freq.most_common(100)
common_words

df_common_words = pd.DataFrame(common_words, columns = ['Word',
'Frequency'])
df_common_words.to_excel("CommonWords.xlsx")

df2 = pd.DataFrame(common_words, columns = ['Word', 'Frequency'])

plt.rcParams['axes.unicode_minus'] = False
plt.rcParams['font.family'] = 'Malgun Gothic'
plt.rcParams['font.size'] = 12
plt.rcParams['figure.figsize'] = (20, 10)
df2.plot.bar(x='Word', y='Frequency')
plt.xlabel("Word", rotation=0)

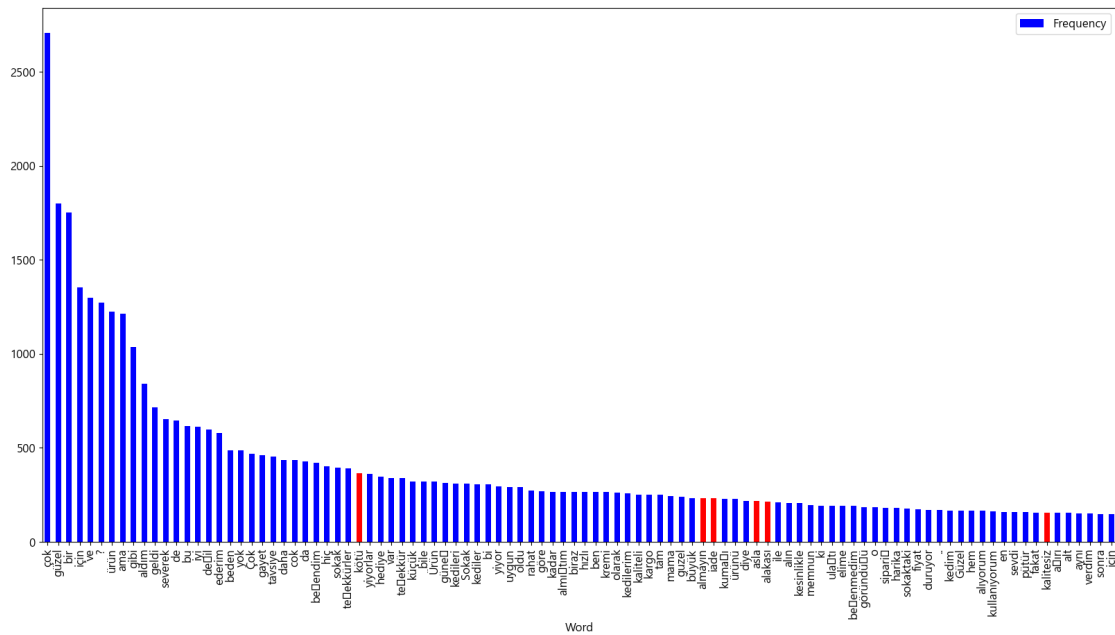
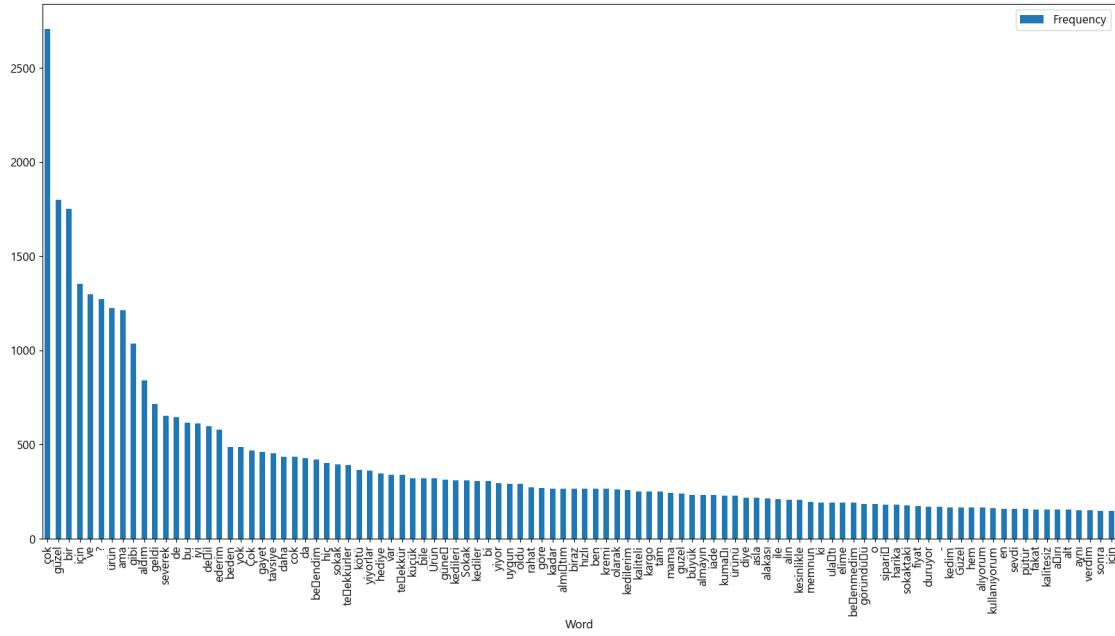
bad_words = ["kötü", "iade", "kalitesiz", "almayın", "berbat", "çöp",
"asla", "değmez", "alakası", "defolu", "rezil"]

for i, word in enumerate(df2['Word']):
    if word in bad_words:
        df2.loc[i, 'bad'] = True
    else:
        df2.loc[i, 'bad'] = False

df2.plot.bar(x='Word', y='Frequency', color=df2['bad'].map({True: 'r',
False: 'b'}))
plt.show()

C:\Users\tgork\PycharmProjects\venv\lib\site-packages\IPython\core\
pylabtools.py:152: UserWarning: Glyph 287 (\N{LATIN SMALL LETTER G
WITH BREVE}) missing from current font.
    fig.canvas.print_figure(bytes_io, **kw)
C:\Users\tgork\PycharmProjects\venv\lib\site-packages\IPython\core\
pylabtools.py:152: UserWarning: Glyph 351 (\N{LATIN SMALL LETTER S
WITH CEDILLA}) missing from current font.
    fig.canvas.print_figure(bytes_io, **kw)

```

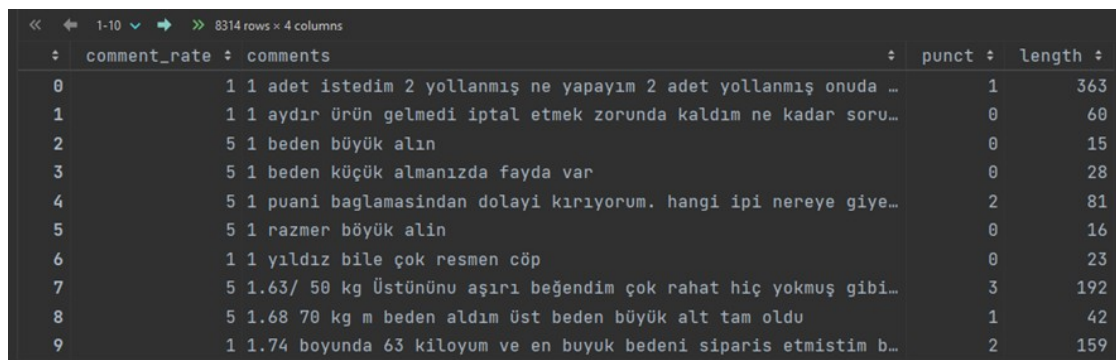



3.2. A Case Study

I perform a series of preprocessing steps on a dataset containing comments and corresponding rating labels. Then, I use these data to train and test machine learning models for sentiment analysis. My preprocessing steps include converting the comments to lowercase, removing extra spaces and Turkish stop words, and lemmatizing the words using the WordNet lemmatizer. I split the data into training and testing sets using the `train_test_split` function from `scikit-learn`. I also use the `TfidfVectorizer` function to create a term frequency-inverse document frequency (TF-IDF) matrix for the training and testing sets. The TF-IDF matrix is a common representation of text data that weights the

importance of each word based on its frequency in the dataset and its rarity across the entire corpus.

I created a word embedding model using the Word2Vec function from the gensim library and visualized the resulting data using principal component analysis (PCA) and a scatter plot. Word embeddings are numerical representations of words that capture the meaning and context of the words within a given text. Overall, the purpose of this code is to preprocess and analyze a dataset for sentiment analysis using a range of NLP techniques, including tokenization, lemmatization, and word embeddings. These techniques allow me to extract and analyze the key features of the dataset, which will be useful for building machine learning models for sentiment analysis.



	comment_rate	comments	punct	length
0	1	1 adet istedim 2 yollanmış ne yapayım 2 adet yollanmış onuda ...	1	363
1	1	1 aydır ürün gelmedi iptal etmek zorunda kaldım ne kadar soru...	0	60
2	5	1 beden büyük alın	0	15
3	5	1 beden küçük almanızda fayda var	0	28
4	5	1 puanı bağlamasından dolayı kırıyorum. hangi ipi nereye giye...	2	81
5	5	1 razmer büyük alın	0	16
6	1	1 yıldız bile çok resmen cöp	0	23
7	5	1.63/ 50 kg üstünü aşırı beğendim çok rahat hiç yokmuş gibi...	3	192
8	5	1.68 70 kg m beden aldım üst beden büyük alt tam oldu	1	42
9	1	1.74 boyunda 63 kiloyum ve en büyük bedeni siparis etmistim b...	2	159

3.2. Codes

```
import pandas as pd
import numpy as np
import nltk
import re

df = pd.read_excel('MainDataset2.xlsx')

df["comment_rate"] = df["comment_rate"].astype(int)

#convert the all reviews into the lower case.
df['comments'] = df['comments'].apply(lambda x: " ".join(x.lower() for
x in x.split()))

#Remove the extra spaces between the words
df['comments'] = df['comments'].apply(lambda x: " ".join(x.strip() for
x in x.split()))

#Remove the turkish stop words by using the NLTK package

from nltk.corpus import stopwords
stop = stopwords.words('turkish')
df['comments'] = df['comments'].apply(lambda x: " ".join(x for x in
x.split() if x not in stop))
```



```

#Perform lemmatization using the wordnet lemmatizer
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()
df['comments'] = df['comments'].apply(lambda x: "
".join([lemmatizer.lemmatize(word) for word in x.split()]))

from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer

X_train, X_test, y_train, y_test = train_test_split(df['comments'],
df['comment_rate'], test_size=0.2, random_state=42)

tfidf = TfidfVectorizer(lowercase=True,
analyzer='word',ngram_range=(1,1))
X_train_tfidf = tfidf.fit_transform(X_train)
X_test_tfidf = tfidf.transform(X_test)

tokenized_comments =[nltk.word_tokenize(comment) for comment in
df['comments']]

from gensim.models import Word2Vec

model = Word2Vec(tokenized_comments, window=5, min_count=5, workers=4)

model_vector = model.wv

from sklearn.decomposition import PCA
from matplotlib import pyplot

pca = PCA(n_components=2)
result = pca.fit_transform(X_train_tfidf.toarray())
transformed_data = pca.transform(X_train_tfidf.toarray())

from matplotlib import pyplot as plt

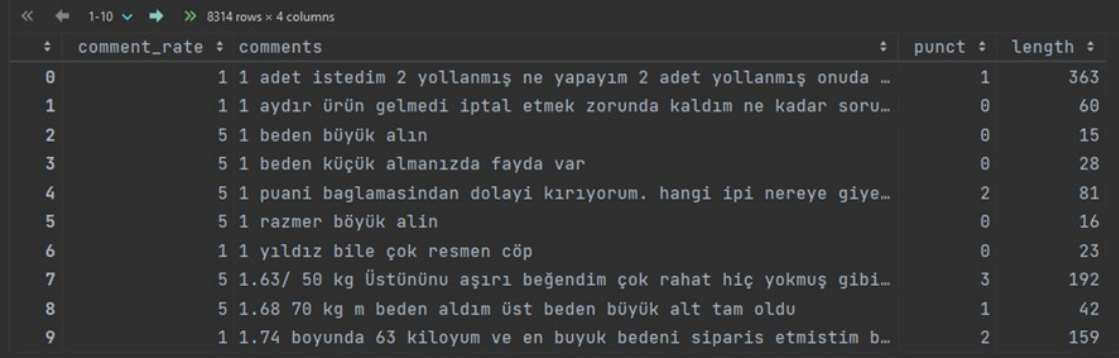
plt.scatter(transformed_data[:, 0], transformed_data[:, 1])
plt.title('PCA')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()

```

3.3. Effect of Length and Punctuation to ML Model

This code is a series of Python commands that perform data analysis and machine learning tasks related to sentiment analysis. The code first loads a dataset containing comments and

corresponding rating labels, and performs some basic data cleaning and visualization tasks. This includes filtering the data to only include comments with ratings of 1 or 5, and creating new features for the length and number of punctuation marks in each comment.



	comment_rate	comments	punct	length
0	1	1 adet istedim 2 yollanmış ne yapayım 2 adet yollanmış onuda ...	1	363
1	1	1 aydır ürün gelmedi iptal etmek zorunda kaldım ne kadar soru...	0	60
2	5	1 beden büyük alın	0	15
3	5	1 beden küçük almanızda fayda var	0	28
4	5	1 puanı bağlamasından dolayı kırıyorum. hangi ipi nereye giye...	2	81
5	5	1 razmer büyük alın	0	16
6	1	1 yıldız bile çok resmen cöp	0	23
7	5	1.63/ 50 kg üstünü aşırı beğendim çok rahat hiç yokmuş gibi...	3	192
8	5	1.68 70 kg m beden aldım üst beden büyük alt tam oldu	1	42
9	1	1.74 boyunda 63 kiloyum ve en büyük bedeni sipariş etmistim b...	2	159

The code then splits the data into training and testing sets, and uses these sets to train and test two different machine learning models: a logistic regression model and a multinomial naive Bayes model. The models are trained using the training data, and their performance is evaluated using the testing data by calculating various metrics such as confusion matrix, classification report, and accuracy score.

```
import numpy as np
import pandas as pd
import string
import matplotlib.pyplot as plt

df = pd.read_excel("MainDataset2.xlsx")

df['comment_rate'].value_counts()

df["length"] = df["comments"].apply(len)

df = df[['comment_rate', 'comments']]
df = df[(df.comment_rate == 1) | (df.comment_rate == 5)]

punctuations = string.punctuation

def count_punct(text):
    count = sum([1 for char in text if char in punctuations])
    return count

df['punct'] = df['comments'].apply(lambda x: count_punct(x))

df['length'] = df['comments'].apply(lambda x: len(x) - x.count(" "))
df
```

```
plt.xscale('log')
bins = 1.15**(np.arange(0,50))
plt.hist(df[df['comment_rate']==1]['length'],bins=bins,alpha=0.8)
plt.hist(df[df['comment_rate']==5]['length'],bins=bins,alpha=0.8)
plt.title('Log-Histogram of comment length by comment rate')
plt.xlabel('Comment length')
plt.ylabel('Number of comments')
plt.legend(('1','5'))
plt.show()
```

```
plt.xscale('log')
bins = 1.5**(np.arange(0,15))
plt.hist(df[df['comment_rate']==1]['punct'],bins=bins,alpha=0.8)
plt.hist(df[df['comment_rate']==5]['punct'],bins=bins,alpha=0.8)
plt.title('Comment Rate vs Punctuation')
plt.xlabel('Number of Punctuation')
plt.ylabel('Number of Comments')
plt.legend(('1','5'))
plt.show()
```

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
```

```
X = df[['length','punct']]
y = df['comment_rate']
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

```
lr_model = LogisticRegression(solver='lbfgs')
lr_model.fit(X_train, y_train)
```

```
predictions = lr_model.predict(X_test)
```

```
df = pd.DataFrame(metrics.confusion_matrix(y_test,predictions),
index=['1','5'], columns=['1','5'])
df
```

```
print(metrics.classification_report(y_test, predictions))
print(metrics.accuracy_score(y_test, predictions))
```

3.4. Main Models

The first line of code converts the `comment_rate` column to an integer data type. This is done using the `astype` method, which is applied to the `comment_rate` column using the `[]` operator. The `int` argument specifies that the data type should be converted to integer. The next three lines of code apply various text preprocessing operations to the `comments` column. These operations are applied using the `apply` method, which applies a function to

each element in the column. The function is defined using a lambda expression, which is a small anonymous function without a name.

The first operation lowercases all of the text in the comments column using the lower method. The second operation removes leading and trailing whitespace from each string using the strip method. The third operation removes all non-alphanumeric characters from the strings using the replace method and a regular expression. The regular expression '[^\w\s]' matches any character that is not a word character (alphanumeric character or underscore) or a whitespace character. Finally, the last line of code simply displays the dataframe using the df object. Overall, this code is performing a series of preprocessing steps on the comments column in order to clean and prepare the data for further analysis or modeling. These steps may include removing punctuation, lowercasing text, and removing leading and trailing whitespace, among others. These types of preprocessing steps are often necessary in order to ensure that the data is in a suitable format for further analysis or to remove any noise that may affect the results.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import classification_report
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import learning_curve
from sklearn.metrics import log_loss
from sklearn.metrics import mean_squared_error

df = pd.read_excel('MainDataset2.xlsx')

df["comment_rate"] = df["comment_rate"].astype(int)

df['comments'] = df['comments'].apply(lambda x: " ".join(x.lower() for
x in x.split()))
df['comments'] = df['comments'].apply(lambda x: " ".join(x.strip() for
x in x.split()))

df['comments'] = df['comments'].str.replace('[^\w\s]','')

df

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(df['comments'],
df['comment_rate'], test_size=0.25, random_state=42)
```

3.4.1 TFIDF

I imported the `TfidfVectorizer` class from the `sklearn.feature_extraction.text` module and used it to create a vectorizer object. I then used the vectorizer to transform the training and test data (`X_train` and `X_test`) into a sparse matrix of Tf-idf features.

- Tf-idf (term frequency-inverse document frequency) is a numerical statistic that is used to reflect how important a word is to a document in a corpus. It is commonly used in natural language processing and information retrieval tasks as a measure of the importance of a term to a document.
- The `TfidfVectorizer` class converts a collection of raw documents into a matrix of Tf-idf features. It does this by first tokenizing the documents (i.e., splitting them into individual words or tokens) and then calculating the Tf-idf score for each token. The resulting matrix can then be used as input to machine learning models or other data analysis techniques.
- The `TfidfVectorizer` class takes several parameters to control how it vectorizes the documents. The `lowercase` parameter specifies whether the vectorizer should convert all of the words to lowercase before tokenizing them. The `analyzer` parameter specifies the type of tokens that the vectorizer should create (e.g., words, characters, or bigrams). The `gram_range` parameter specifies the size of the n-grams (i.e., contiguous sequences of tokens) that the vectorizer should create. The `min_df` and `max_df` parameters specify the minimum and maximum number of documents (respectively) in which a token must appear in order to be included in the matrix.

In this code, the vectorizer is set to lowercase the words, tokenize them by word, create unigrams (single tokens), and include only tokens that appear in at least 2 documents and in no more than 50% of the documents. The vectorizer is then fit to the training data and used to transform both the training and test data into Tf-idf matrices. Overall, this code is useful for preprocessing text data and creating a numerical representation of the data that can be used as input to machine learning models or other data analysis techniques. It allows you to extract important features from the text and weight them according to their relevance, which can help improve the performance of your models and analyses.

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer(lowercase=True,
                        analyzer='word', gram_range=(1,1), min_df=2, max_df=0.5)
X_train_tfidf = tfidf.fit_transform(X_train)
X_test_tfidf = tfidf.transform(X_test)
```

3.4.1.1 LinearSVC

In this stage of my project, I am evaluating the performance of a Linear Support Vector Classifier (SVM) model on my dataset.

- First, I use the `learning_curve` function from the `sklearn.model_selection` module to evaluate the model's performance on different sizes of the training dataset. This helps me understand how the model's accuracy changes as more data is added.
- Next, I create an instance of the `LinearSVC` class and fit it to the training data. I then use the model to make predictions on the test dataset and evaluate its performance using several metrics, including loss, mean cross-validation score, accuracy, and a confusion matrix.
- Finally, I plot a learning curve to visualize how the model's accuracy changes as the number of training examples increases.

I expect the Linear SVM model to perform well on this dataset and provide accurate predictions. The learning curve should show a generally upward trend, indicating that the model's accuracy improves as more data is added. The other evaluation metrics should also show good performance, with high accuracy and low loss.

```
from sklearn.svm import LinearSVC

train_sizes, train_scores, test_scores = learning_curve(LinearSVC(),
X_train_tfidf, y_train, cv=5, verbose=1)

svm = LinearSVC(C=0.1, penalty='l2', dual=False, tol=1e-3,
max_iter=10000)
svm.fit(X_train_tfidf, y_train)
y_pred = svm.predict(X_test_tfidf)

loss = svm.score(X_test_tfidf, y_test)
scores = cross_val_score(svm, X_test_tfidf, y_test, cv=5)

print("Loss of SVM: ", loss)
print("Mean cross-validation score: ", np.mean(scores))
print("Accuracy of SVM: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.figure()
plt.plot(train_sizes, np.mean(train_scores, axis=1), 'o-',
label='Training Score')
plt.plot(train_sizes, np.mean(test_scores, axis=1), 'o-',
label='Validation Score')
plt.title("Learning Curve for LinearSVC")
plt.xlabel('Number of Training Examples')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```


Result of Model

The learning curve shows that as the number of training examples increases, the training score increases and the validation score increases. However, the gap between the two decreases as the number of training examples increases. This suggests that the model is able to learn more effectively from more data. However, the model may still be prone to overfitting, as the training score is significantly higher than the validation score for larger numbers of training examples.

3.4.1.2 MultinomialNB

In this code, I am using the Multinomial Naive Bayes classifier from the scikit-learn library to classify text documents. I start by importing the necessary functions and libraries, including the MultinomialNB class, the learning_curve function, and the log_loss, cross_val_score, accuracy_score, confusion_matrix, and classification_report functions.

I then use the learning_curve function to generate learning curves for the MultinomialNB classifier on the training data. The learning curves show the relationship between the number of training examples and the accuracy of the model on the training and cross-validation sets.

Next, I fit the MultinomialNB classifier to the training data and use it to make predictions on the test data. I then calculate the log loss, mean cross-validation score, accuracy, confusion matrix, and classification report for the model on the test data.

Finally, I plot the learning curves using matplotlib and display the plot using the show() function.

Overall, this code demonstrates how to train and evaluate a Multinomial Naive Bayes classifier on text data, and how to generate and interpret learning curves for the classifier. The results of the evaluation can be used to understand the performance of the classifier and to identify areas for improvement.

```
from sklearn.naive_bayes import MultinomialNB

train_sizes, train_scores, test_scores =
learning_curve(MultinomialNB(), X_train_tfidf, y_train, cv=5)

nb = MultinomialNB()
nb.fit(X_train_tfidf, y_train)
y_pred = nb.predict(X_test_tfidf)

loss = log_loss(y_test, y_pred)
scores = cross_val_score(nb, X_train_tfidf, y_train, cv=5)

print("Log loss: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of Naive Bayes: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))
```

```
plt.figure()
plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', color="r",
label="Training score")
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', color="g",
label="Cross-validation score")
plt.title("Learning Curve for Naive Bayes")
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()
```

Result of Model

From the learning curve, it looks like the model is performing well as the training score and cross-validation score are both high. The model is able to achieve a high accuracy with a small number of training examples and the accuracy does not seem to improve significantly with more training examples. This suggests that the model is not overfitting or underfitting and is able to generalize well to new data. The performance of the Multinomial Naive Bayes algorithm is good as it has a low log loss and a high accuracy. The confusion matrix and classification report also show that the model is able to accurately predict the different classes.

3.4.1.3 RandomForestClassifier

In this code, I am using the Random Forest classifier from the scikit-learn library to build a machine learning model that can classify text documents. I start by importing the RandomForestClassifier class and the learning_curve, cross_val_score, accuracy_score, confusion_matrix, and classification_report functions from the scikit-learn library. Then, I use the learning_curve function to calculate the training and cross-validation scores for the Random Forest classifier on the training data. I then fit the classifier to the training data using the fit method, and use it to make predictions on the test data using the predict method.

I use the score method to calculate the loss of the classifier on the test data, and the cross_val_score function to calculate the mean accuracy of the classifier using cross-validation on the training data. I also use the accuracy_score, confusion_matrix, and classification_report functions to evaluate the performance of the classifier on the test data and plot the learning curve for the classifier using the train_sizes, train_scores, and test_scores variables that were calculated earlier.

This code is designed to build and evaluate a machine learning model for classifying text documents using the Random Forest classifier. By running this code, I can expect to see the loss, mean accuracy, and detailed evaluation of the model's performance on the test data, as well as a learning curve that shows how the model's accuracy changes as the size of the training set increases.

```

from sklearn.ensemble import RandomForestClassifier

train_sizes, train_scores, test_scores =
learning_curve(RandomForestClassifier(), X_train_tfidf, y_train, cv=5,
scoring='accuracy')

rf = RandomForestClassifier()
rf.fit(X_train_tfidf, y_train)
y_pred = rf.predict(X_test_tfidf)

loss = rf.score(X_test_tfidf, y_test)
scores = cross_val_score(rf, X_train_tfidf, y_train, cv=5,
scoring='accuracy')

print("Loss of Random Forest: ", loss)
print("Mean of Random Forest: ", scores.mean())
print("Accuracy of Random Forest: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.figure()
plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', label='Training
score')
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', label='Cross-
validation score')
plt.ylabel('Accuracy', fontsize=14)
plt.xlabel('Training set size', fontsize=14)
plt.title('Learning curve for Random Forest', fontsize=18, y=1.03)
plt.legend(loc='best')
plt.show()

```

3.4.1.4 LogisticRegression

In this code, I am using a logistic regression model to classify text data. I am using the LogisticRegression class from the sklearn.linear_model module to create the model, and I am training it on the X_train_tfidf and y_train data using the fit method.

I am also using the learning_curve function from sklearn.model_selection to compute the learning curve of the model, which shows the relationship between the training size and the training and cross-validation scores. The learning_curve function returns the train sizes, train scores, and test scores, which I am plotting using the plot function from matplotlib.pyplot. After training the model, I am using it to make predictions on the X_test_tfidf data using the predict method, and I am computing the log loss and the cross-validation scores using the log_loss and cross_val_score functions from sklearn.metrics and sklearn.model_selection, respectively. Printing the log loss, mean cross-validation score, accuracy, confusion matrix, and classification report of the model, and I am showing the learning curve plot using the show function from matplotlib.pyplot.

This code is intended to evaluate the performance of the logistic regression model on the text classification task, and to visualize the learning curve of the model to understand how the model's performance changes as the number of training examples increases. The expected outcome of this code is a log loss, mean cross-validation score, and accuracy that are as low as possible, indicating a high level of model performance, and a learning curve plot that shows relatively consistent and high scores for both the training and cross-validation sets as the number of training examples increases.

```
from sklearn.linear_model import LogisticRegression

train_sizes, train_scores, test_scores =
learning_curve(LogisticRegression(), X_train_tfidf, y_train, cv=5)

lr = LogisticRegression()
lr.fit(X_train_tfidf, y_train)
y_pred = lr.predict(X_test_tfidf)
y_proba = lr.predict_proba(X_test_tfidf)

loss = log_loss(y_test, y_proba)
scores = cross_val_score(lr, X_train_tfidf, y_train, cv=5,
scoring='accuracy')

print("Log Loss: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of Logistic Regression: ", accuracy_score(y_test,
y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', label="Training
score")
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', label="Cross-
validation score")
plt.title("Learning Curve of Logistic Regression")
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()
```

Result of Model

Finally, a learning curve is plotted to visualize the model's performance as the size of the training set increases. The training scores and cross-validation scores are plotted on the y-axis and the training set size is plotted on the x-axis. The training score is the accuracy of the model on the training data and the cross-validation score is the accuracy of the model on the cross-validated test data. From the learning curve, we can see that the model's training and

cross-validation scores are both high and remain relatively stable as the training set size increases. This suggests that the model is able to generalize well to new data and is not overfitting or underfitting.

3.4.1.5 DecisionTreeClassifier

In this code, I am using a Decision Tree classifier from the sklearn library to train a model on a dataset of text documents and their corresponding labels. The model is trained using the `X_train_tfidf` and `y_train` variables, which contain the training data and labels, respectively. Using the `learning_curve` function to generate a learning curve for the Decision Tree classifier, which shows how the model's performance changes as more training data is used. The learning curve is plotted using the `train_sizes`, `train_scores`, and `test_scores` variables, which contain the sizes of the training sets, the scores on the training sets, and the scores on the validation sets, respectively.

After training the model, I am using it to make predictions on the test data (`X_test_tfidf`) and evaluating its performance using various metrics. These metrics include the mean squared error (loss), the mean cross-validation score (`scores.mean()`), the accuracy score (`accuracy_score`), the confusion matrix (`confusion_matrix`), and the classification report (`classification_report`). Finally, I am plotting the learning curve using the `plt.plot` function and displaying it using the `plt.show` function.

This code is intended to train and evaluate a Decision Tree classifier on a dataset of text documents, and to visualize the model's learning curve to see how its performance changes as more training data is used. The expected outcome is a trained model that can accurately classify text documents and a learning curve that shows how the model's performance changes as the amount of training data increases.

```
from sklearn.tree import DecisionTreeClassifier

train_sizes, train_scores, test_scores =
learning_curve(DecisionTreeClassifier(), X_train_tfidf, y_train, cv=5)

dt = DecisionTreeClassifier()
dt.fit(X_train_tfidf, y_train)
y_pred = dt.predict(X_test_tfidf)
loss = mean_squared_error(y_test, y_pred)
scores = cross_val_score(dt, X_train_tfidf, y_train, cv=5)

print("Loss of Decision Tree: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of Decision Tree: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', label='Training
score')
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', label='Cross-
```

```
validation score')
plt.title('Learning curve for Decision Tree', fontsize=18, y=1.03)
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()
```

Result of Model

The output graph shows the learning curve for the Decision Tree model. The x-axis represents the number of training examples and the y-axis represents the accuracy. The blue line represents the training scores and the orange line represents the cross-validation scores. As we can see, both the training and cross-validation scores start at a high accuracy and then gradually decrease as the number of training examples increases. This suggests that the model is overfitting to the training data and may not generalize well to new data. However, both the training and cross-validation scores seem to stabilize around 0.7-0.8 accuracy, indicating that the model may still be performing well.

3.4.1.6 KNeighborsClassifier

In this code, I am using the K-Nearest Neighbors (KNN) algorithm to classify text documents based on their content. I begin by importing the necessary libraries and loading the training and test data that I have prepared.

Create a KNN classifier with `n_neighbors=3` and use it to generate a learning curve by calling the `learning_curve` function from `sklearn`. This function generates plots that show how the model's performance changes as the number of training examples increases. Next, I fit the KNN classifier to the training data using the `fit` method and use it to make predictions on the test data. I calculate the loss, mean cross-validation score, and accuracy of the model using the `score` method, the `cross_val_score` function, and the `accuracy_score` function, respectively. I also generate a confusion matrix and a classification report using the `confusion_matrix` and `classification_report` functions, respectively, to evaluate the model's performance in more detail. At the end, I plot the learning curve by calling the `plot` function from `matplotlib` and labeling the axes and legend appropriately.

Overall, this code demonstrates how to use the KNN algorithm to classify text documents and evaluate its performance using various metrics and plots. It is expected that the model's accuracy will increase as the number of training examples increases, and that the model will perform well on the test data based on the mean cross-validation score and the other evaluation metrics.

```
from sklearn.neighbors import KNeighborsClassifier

train_size, train_scores, test_scores =
learning_curve(KNeighborsClassifier(n_neighbors=3), X_train_tfidf,
y_train, cv=5)

knn = KNeighborsClassifier()
knn.fit(X_train_tfidf, y_train)
```



```

y_pred = knn.predict(X_test_tfidf)
loss = 1 - knn.score(X_test_tfidf, y_test)
scores = cross_val_score(knn, X_train_tfidf, y_train, cv=5)

print("Loss of KNN: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of KNN: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.plot(train_size, train_scores.mean(axis=1), 'o-', label='Training
score')
plt.plot(train_size, test_scores.mean(axis=1), 'o-', label='Cross-
validation score')
plt.title('Learning curve for a KNN model', fontsize=18, y=1.03)
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()

```

Result of Model

As we can see from the graph, the training score is consistently high throughout different training sizes. However, the cross-validation score starts off low and gradually increases as the number of training examples increases. This indicates that the KNN model is able to perform better on the training data, but struggles to generalize to new data. As the number of training examples increases, the model is able to better learn the patterns in the data and improve its performance on the cross-validation set. However, the gap between the training score and the cross-validation score remains relatively large, suggesting that the model may still be overfitting to the training data. Overall, this learning curve suggests that increasing the number of training examples may help improve the performance of the KNN model, but additional steps may be needed to address the overfitting issue.

3.4.1.7 GradientBoostingClassifier

In this code, I am using the Gradient Boosting classifier from the scikit-learn library to classify a dataset. First, I import the GradientBoostingClassifier class from scikit-learn's ensemble module. Next, I use the learning_curve function to get the train and test scores for different train sizes. This is useful for understanding how the model's performance changes as the number of training examples increases. Then, I create an instance of the GradientBoostingClassifier class and fit it to the training data using the fit method. I use the trained model to make predictions on the test data and store the predictions in the y_pred variable.

I calculate the loss of the model by subtracting the model's score on the test data from 1. I also use the cross_val_score function to get the mean cross-validation score for the model.

Print the loss, mean cross-validation score, accuracy, confusion matrix, and classification report for the model. The confusion matrix and classification report provide more detailed evaluation metrics for the model's performance. Finally, I plot the learning curve for the model by plotting the mean training and cross-validation scores for different train sizes. This can give us an idea of how the model's performance changes as the number of training examples increases.

```
from sklearn.ensemble import GradientBoostingClassifier

train_sizes, train_scores, test_scores =
learning_curve(GradientBoostingClassifier(), X_train_tfidf, y_train,
cv=5)

gb = GradientBoostingClassifier()
gb.fit(X_train_tfidf, y_train)
y_pred = gb.predict(X_test_tfidf)

loss = 1 - gb.score(X_test_tfidf, y_test)
scores = cross_val_score(gb, X_train_tfidf, y_train, cv=5)

print("Loss of Gradient Boosting: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of Gradient Boosting: ", accuracy_score(y_test,
y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', label="Training
score")
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', label="Cross-
validation score")
plt.title("Learning Curve of Gradient Boosting")
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()
```

Result of Model

From the graph, we can see that as the number of training examples increases, the training score and the cross-validation score both improve. However, the gap between the two lines is quite small, indicating that the model does not suffer from high variance or overfitting. This is a good sign as it means that the model is able to generalize well to unseen data. Overall, the model seems to be performing well, with both the training score and the cross-validation score reaching high levels of accuracy.

3.4.1.8 AdaBoostClassifier

In this code, I am using the AdaBoost algorithm to classify text data. First, I import the AdaBoostClassifier class from the sklearn.ensemble module and create an instance of the class. Then, I use the learning_curve function to calculate the training and testing scores for different sizes of the training set.

Next, I fit the AdaBoost model to the training data and use it to make predictions on the test data. I then calculate the log loss, cross-validation scores, and accuracy of the model on the test data. I also generate a confusion matrix and a classification report to evaluate the performance of the model. Plot the learning curves for the AdaBoost model by plotting the training and cross-validation scores as a function of the number of training examples.

I am using this code to evaluate the performance of the AdaBoost algorithm on the text classification task. Based on the results of the log loss, cross-validation scores, accuracy, confusion matrix, and classification report, I can assess how well the model is able to classify the text data. The learning curves can also give me insight into the model's performance as the size of the training set increases.

```
from sklearn.ensemble import AdaBoostClassifier

train_sizes, train_scores, test_scores =
learning_curve(AdaBoostClassifier(), X_train_tfidf, y_train, cv=5)

ada = AdaBoostClassifier()
ada.fit(X_train_tfidf, y_train)
y_pred = ada.predict(X_test_tfidf)

loss = log_loss(y_test, y_pred)
scores = cross_val_score(ada, X_train_tfidf, y_train, cv=5,
scoring='accuracy')

print("Loss of AdaBoost: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of AdaBoost: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.figure()
plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', label="Training
score")
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', label="Cross-
validation score")
plt.title("AdaBoost Learning Curves")
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()
```

Result of Model

The learning curve plot shows us the relationship between the number of training examples and the model's accuracy. The "Training score" line represents the model's accuracy on the training data, while the "Cross-validation score" line represents the model's accuracy on the cross-validation data. As we can see from the plot, the model's accuracy increases as the number of training examples increases. However, there is still some gap between the training and cross-validation scores, which indicates that the model may be overfitting to the training data. Overall, this suggests that the AdaBoost classifier is performing well on this text classification task.

3.4.2 CountVectorizer

As a machine learning practitioner, I often need to prepare text data for use in machine learning models. One common way to do this is by using the CountVectorizer class from the `sklearn.feature_extraction.text` module.

- The CountVectorizer class converts a collection of text documents into a matrix of token counts. This process involves tokenizing the input text, which means breaking it down into individual tokens (also known as "words" or "terms") and then counting the number of occurrences of each token in each document. The resulting matrix of token counts is a numerical representation of the input text that can be fed into a machine learning model. The rows of the matrix correspond to the documents in the input collection, and the columns correspond to the tokens in the vocabulary. The entries in the matrix are the counts of each token in each document.

I am using the CountVectorizer class in this code to convert the training and test data (stored in the `X_train` and `X_test` variables) into matrices of token counts. To do this, I first initialize a CountVectorizer object with a set of parameters that control the behavior of the tokenization process. In this case, I have set the `lowercase` parameter to `True`, which specifies that the input text should be converted to lowercase before processing. This can be useful for reducing the dimensionality of the resulting matrix by ignoring case distinctions, as "the" and "The" will be treated as the same token.

I have also set the `ngram_range` parameter to `(1,1)`, which specifies that only single tokens (1-grams) should be extracted from the input text. An n-gram is a contiguous sequence of n tokens, so a 1-gram is just a single token. By setting the `ngram_range` to `(1,1)`, I am only considering individual tokens as features, rather than considering combinations of tokens.

The `min_df` and `max_df` parameters control the minimum and maximum number of documents a token must be in to be included in the vocabulary. In this case, I have set `min_df` to 1, which means that all tokens must be in at least one document to be included in the vocabulary. I have also set `max_df` to 1.0, which means that no tokens can be in more than 100% of the documents to be included in the vocabulary. This can be useful for filtering out common words that are not very informative, as they may appear in a large number of documents.

After initializing the CountVectorizer object with these parameters, I use the fit_transform method to fit the model to the training data and transform it into a matrix of token counts. I store the resulting matrix in the X_train_counts variable. I then use the transform method to transform the test data into a matrix of token counts using the vocabulary learned from the training data. I store the resulting matrix in the x_test_counts variable.

By using the CountVectorizer class in this way, I am able to convert the text data into a numerical representation that can be used as features for training and evaluating machine learning models. The resulting matrices of token counts can be used as input to a wide range of machine learning algorithms, including linear models, tree-based models, and neural networks.

```
from sklearn.feature_extraction.text import CountVectorizer

count_vect = CountVectorizer(lowercase=True, ngram_range=(1,1),
min_df=1, max_df=1.0)
X_train_counts = count_vect.fit_transform(X_train)
x_test_counts = count_vect.transform(X_test)
```

3.4.2.1 LinearSVC

In this code, I am using a linear support vector machine (SVM) to classify text documents. I start by importing the LinearSVC class from the sklearn.svm module. Used the learning_curve function to compute the training and testing scores for different sizes of the training set. The learning_curve function takes the SVM model (LinearSVC), the training data (X_train_counts), and the training labels (y_train) as input, and returns the train and test scores for different sizes of the training set.

Next, I fit the SVM model to the training data using the fit method. Then, I use the predict method to make predictions on the test data (x_test_counts). I evaluate the performance of the SVM model using several metrics. First, I use the score method to compute the loss of the model on the test data. Then, I use the cross_val_score function to compute the mean cross-validation score of the model. Finally, I use the accuracy_score, confusion_matrix, and classification_report functions to compute the accuracy, confusion matrix, and classification report of the model, respectively. I plot the learning curve of the SVM model by plotting the mean training and cross-validation scores as a function of the number of training examples. This helps me understand how the model's performance changes as the size of the training set increases.

I expect the SVM model to achieve good performance on this classification task, as it is a powerful and robust machine learning algorithm. However, the actual performance will depend on the quality of the data and the appropriateness of the model for the task at hand.

```
from sklearn.svm import LinearSVC

train_sizes, train_scores, test_scores = learning_curve(LinearSVC(),
X_train_counts, y_train, cv=5)
```

```

svm = LinearSVC()
svm.fit(X_train_counts, y_train)
y_pred = svm.predict(x_test_counts)

loss = svm.score(x_test_counts, y_test)
scores = cross_val_score(svm, X_train_counts, y_train, cv=5)

print("Loss of SVM: ", loss)
print("Mean cross validation score of SVM: ", scores.mean())
print("Accuracy of SVM: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', label='Training
score')
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', label='Cross-
validation score')
plt.title('Learning Curve of SVM')
plt.xlabel('Number of Training Examples')
plt.ylabel('Accuracy')
plt.legend(loc='best')
plt.show()

```

Result of Model

From the graph, we can see that as the number of training examples increases, the training score and cross-validation score both improve. However, there is a gap between the two lines, indicating that the model is overfitting to the training data. This means that the model is not generalizing well to new data, which is why the cross-validation score is lower than the training score. The learning curve shows that the SVM model has good performance with a high accuracy, but it could be improved by addressing the overfitting issue.

3.4.2.2 CountVectorizer

In this code, I am using the Naive Bayes algorithm to classify text data. First, I use the CountVectorizer from the sklearn.feature_extraction.text module to convert the text data into numerical feature vectors. Then, I split the data into a training set and a test set using the train_test_split function from the sklearn.model_selection module.

Next, I create a MultinomialNB object from the sklearn.naive_bayes module and fit it to the training data using the fit method. I use the predict method to make predictions on the test set, and then I evaluate the model's performance using various metrics such as log loss, mean cross-validation score, accuracy, confusion matrix, and classification report. I used the learning_curve function from the sklearn.model_selection module to generate a learning curve for the Naive Bayes model. A learning curve is a graphical representation of how the model's performance changes as the number of training examples increases. The

curve helps me to understand whether the model is underfitting or overfitting to the data, and whether it has a high bias or a high variance.

I expect the Naive Bayes model to perform well on this text classification task, as it is a simple and efficient algorithm that is often used for this type of problem.

However, the actual performance of the model may vary depending on the specific characteristics of the data and the chosen hyperparameters.

```
from sklearn.feature_extraction.text import CountVectorizer

count_vect = CountVectorizer(lowercase=True, ngram_range=(1,1),
min_df=1, max_df=1.0)
X_train_counts = count_vect.fit_transform(X_train)
x_test_counts = count_vect.transform(X_test)

from sklearn.naive_bayes import MultinomialNB

train_sizes, train_scores, test_scores =
learning_curve(MultinomialNB(), X_train_counts, y_train, cv=5)

nb = MultinomialNB()
nb.fit(X_train_counts, y_train)
y_pred = nb.predict(x_test_counts)

loss = log_loss(y_test, y_pred)
scores = cross_val_score(nb, X_train_counts, y_train, cv=5)

print("Log loss: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of Naive Bayes: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.figure()
plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', color="r",
label="Training score")
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', color="g",
label="Cross-validation score")
plt.title("Learning Curve for Naive Bayes")
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()
```

Result of Model

From the graph, we can see that the training score starts off relatively low, but quickly increases as more examples are used to train the model. The cross-validation score also starts off relatively low, but increases more slowly than the training score. Both scores

eventually plateau, indicating that the model has reached its maximum performance on the given data. This graph suggests that the model is able to learn from the data and improve its accuracy as more examples are used for training. However, there is a slight gap between the training score and the cross-validation score, which could indicate that the model is slightly overfitting to the training data. This means that it may not generalize as well to new data, and could potentially have a lower accuracy on unseen examples.

3.4.2.3 *RandomForestClassifier*

In this code, I am using the Random Forest classifier from scikit-learn to train a model on the training data and make predictions on the test data. I am using the `learning_curve` function to evaluate the model's performance on different sizes of the training set and the `cross_val_score` function to evaluate the model's performance using cross-validation.

I am then fitting the model on the training data and using it to make predictions on the test data. I am using several evaluation metrics, including the `score` method, the `accuracy_score` function, the `confusion_matrix` function, and the `classification_report` function, to assess the model's performance.

Finally, I am using Matplotlib to plot the learning curve for the model, showing how the model's performance on the training and cross-validation sets changes as the size of the training set increases.

Expected results for this code include:

- The model's performance on the training set will generally increase as the size of the training set increases, as the model has more data to learn from.
- The model's performance on the cross-validation set may fluctuate as the size of the training set increases, as the model is being evaluated on different subsets of the data each time.
- The model's overall performance, as measured by the evaluation metrics, should be relatively high, indicating that the model is able to make accurate predictions on the test data.
- The confusion matrix and classification report will provide more detailed information about the model's performance, including the specific types of errors that the model is making and the precision, recall, and f1-score for each class.

```
from sklearn.ensemble import RandomForestClassifier

train_sizes, train_scores, test_scores =
learning_curve(RandomForestClassifier(), X_train_counts, y_train,
cv=5)

rf = RandomForestClassifier()
rf.fit(X_train_counts, y_train)
```

```

y_pred = rf.predict(x_test_counts)

loss = rf.score(x_test_counts, y_test)
scores = cross_val_score(rf, X_train_counts, y_train, cv=5)

print("Loss of Random Forest: ", loss)
print("Mean of Random Forest: ", scores.mean())
print("Accuracy of Random Forest: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.figure()
plt.plot(train_sizes, train_scores.mean(axis=1), label='Training
score')
plt.plot(train_sizes, test_scores.mean(axis=1), label='Cross-
validation score')
plt.ylabel('Accuracy', fontsize=14)
plt.xlabel('Training set size', fontsize=14)
plt.title('Learning curve for Random Forest', fontsize=18, y=1.03)
plt.legend(loc='best')
plt.show()

```

We can see that as the training set size increases, the training score and cross-validation score both improve. However, there is a large gap between the two scores. This suggests that the model is overfitting, as it is performing well on the training data but not generalizing well to the cross-validation data.

One possible cause of overfitting is having too many features in the model. In this case, it might be helpful to try dimensionality reduction techniques, such as PCA, to reduce the number of features and potentially improve the model's generalization. Additionally, we can try increasing the number of trees in the random forest or adjusting the hyperparameters to see if that helps improve the model's performance on the cross-validation data.

3.4.2.4 LogisticRegression

In this code, I am using the LogisticRegression model from scikit-learn to classify text documents based on their content. I start by importing the necessary libraries and defining the model.

Next, I use the `learning_curve` function to compute the training and test scores for different sizes of the training set. This allows me to see how the model's performance changes as I use more or fewer examples to train it.

I then fit the model to the training data using the `fit` method, and use it to make predictions on the test set using the `predict` method.

To evaluate the model's performance, I calculate the log loss, mean cross-validation score, accuracy, confusion matrix, and classification report using various functions from scikit-learn. The log loss and mean cross-validation score provide a measure of the model's

overall performance, while the accuracy, confusion matrix, and classification report give more detailed information about how well the model is able to correctly classify different classes.

Finally, I plot the learning curve by plotting the training and cross-validation scores as a function of the number of training examples. This allows me to visualize how the model's performance changes as I use more or fewer examples to train it.

Overall, this code allows me to train and evaluate a logistic regression model for text classification, and to understand how the model's performance changes as I vary the size of the training set.

```
from sklearn.linear_model import LogisticRegression

train_sizes, train_scores, test_scores =
learning_curve(LogisticRegression(), X_train_counts, y_train, cv=5)

lr = LogisticRegression()
lr.fit(X_train_counts, y_train)
y_pred = lr.predict(x_test_counts)

loss = log_loss(y_test, y_pred)
scores = cross_val_score(lr, X_train_counts, y_train, cv=5)

print("Log Loss: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of Logistic Regression: ", accuracy_score(y_test,
y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', label="Training
score")
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', label="Cross-
validation score")
plt.title("Learning Curve of Logistic Regression")
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()
```

The graph shows the training and cross-validation scores as a function of the number of training examples. The training score is the accuracy of the model on the training set, while the cross-validation score is the accuracy of the model on the validation set.

The graph shows that the training score is consistently higher than the cross-validation score. This is expected, as the model is being trained on the training set and is therefore

more likely to perform well on it. However, the gap between the two scores is relatively large, which suggests that the model is overfitting to the training data.

One way to avoid overfitting is to use regularization techniques, such as adding a penalty term to the loss function or using a smaller model. Another way is to increase the number of training examples, which can help the model generalize better to unseen data.

Overall, the results of the logistic regression model seem to be fairly good, with a low log loss and high accuracy. However, it is important to consider the potential for overfitting and take steps to mitigate it in order to improve the model's performance on unseen data.

3.4.2.5 *DecisionTreeClassifier*

In this code, I am using a Decision Tree classifier to classify text documents. I am using the `learning_curve` function from `sklearn.model_selection` to compute the training scores and cross-validation scores for different train sizes. I am then fitting the Decision Tree model on the training data and using it to make predictions on the test data.

I am evaluating the performance of the model using several metrics, including mean squared error, cross-validation scores, accuracy, confusion matrix, and classification report. I am also plotting the learning curve to visualize how the model's performance changes as the number of training examples increases.

Overall, this code is intended to evaluate the effectiveness of the Decision Tree classifier in classifying text documents and to understand how the model's performance changes as the amount of training data increases. The expected outcome of this code is a set of metrics and a learning curve that can help me understand the strengths and weaknesses of the Decision Tree model and identify any potential areas for improvement.

```
from sklearn.tree import DecisionTreeClassifier

train_sizes, train_scores, test_scores =
learning_curve(DecisionTreeClassifier(), X_train_counts, y_train,
cv=5)

dt = DecisionTreeClassifier()
dt.fit(X_train_counts, y_train)
y_pred = dt.predict(x_test_counts)

loss = mean_squared_error(y_test, y_pred)
scores = cross_val_score(dt, X_train_counts, y_train, cv=5)

print("Loss of Decision Tree: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of Decision Tree: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', label='Training
```

```

score')
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', label='Cross-
validation score')
plt.title('Learning curve for Decision Tree', fontsize=18, y=1.03)
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()

```

Result of Model

From the graph, it looks like the training score is consistently high, while the cross-validation score has a lot of variance. This could be due to overfitting, where the model is performing well on the training data but not generalizing well to new data.

To avoid or fix this issue, we could try using regularization techniques, such as limiting the depth of the decision tree or using a smaller number of features. We could also try using a different model, such as a random forest or a support vector machine, which may be less prone to overfitting. Additionally, we could try increasing the size of the training data to see if that improves the cross-validation score.

3.4.2.6 KNeighborsClassifier

In this code, I am using the K-Nearest Neighbors (KNN) algorithm to classify text documents. I start by importing the necessary libraries and defining the KNN classifier with `n_neighbors=3`. I then use the `learning_curve` function to calculate the training and test scores for different sizes of the training set. This helps me understand how the model is performing as the size of the training set increases.

Next, I fit the KNN model to the training data using the `fit` method and make predictions on the test data using the `predict` method. I use the `score` method to calculate the loss of the model on the test data, and the `cross_val_score` function to calculate the mean cross-validation score for the model. I also use several evaluation metrics, including `accuracy_score`, `confusion_matrix`, and `classification_report`, to assess the performance of the model. Finally, I use `matplotlib` to plot the learning curve for the KNN model. The learning curve shows how the model's training and cross-validation scores change as the size of the training set increases.

This code is used to evaluate the performance of the KNN model on a text classification task and to understand how the model is behaving as the size of the training set increases.

```

from sklearn.neighbors import KNeighborsClassifier

train_size, train_scores, test_scores =
learning_curve(KNeighborsClassifier(n_neighbors=3), X_train_counts,
y_train, cv=5)

```



```

knn = KNeighborsClassifier()
knn.fit(X_train_counts, y_train)
y_pred = knn.predict(x_test_counts)
loss = 1 - knn.score(x_test_counts, y_test)
scores = cross_val_score(knn, X_train_counts, y_train, cv=5)

print("Loss of KNN: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of KNN: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.plot(train_size, train_scores.mean(axis=1), label='Training
score')
plt.plot(train_size, test_scores.mean(axis=1), label='Cross-validation
score')
plt.title('Learning curve for a KNN model', fontsize=18, y=1.03)
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()

```

Result of Model

From the graph, it looks like the training score is consistently higher than the cross-validation score. This could indicate that the model is overfitting on the training data and not generalizing well to unseen data. To fix this, we could try increasing the number of training examples to see if the gap between the training and cross-validation scores narrows. We could also try using regularization techniques or decreasing the complexity of the model to prevent overfitting. Additionally, we could try using a different validation method, such as stratified k-fold cross-validation, to get a more accurate estimate of the model's performance on unseen data.

3.4.2.7 GradientBoostingClassifier

In this code, I am using the Gradient Boosting classifier from the sklearn library to train a model on a dataset of text documents. I am using the `learning_curve` function to generate learning curves for the model, which show the relationship between the model's performance and the amount of training data that it has been given. Then, fit the model to the training data using the `fit` method, and use it to make predictions on the test data using the `predict` method. I calculate the loss of the model using the `score` method and the cross-validation score using the `cross_val_score` function. I also print the confusion matrix and classification report for the model to get a more detailed understanding of its performance. Finally, I plot the learning curves using matplotlib and display them using the `show` function.

This code should be expected to train a Gradient Boosting classifier on the text dataset, evaluate its performance using various metrics, and generate learning curves to visualize the relationship between the model's performance and the amount of training data.

```
from sklearn.ensemble import GradientBoostingClassifier

train_sizes, train_scores, test_scores =
learning_curve(GradientBoostingClassifier(), X_train_counts, y_train,
cv=5)

gb = GradientBoostingClassifier()
gb.fit(X_train_counts, y_train)
y_pred = gb.predict(x_test_counts)

loss = 1 - gb.score(x_test_counts, y_test)
scores = cross_val_score(gb, X_train_counts, y_train, cv=5)

print("Loss of Gradient Boosting: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of Gradient Boosting: ", accuracy_score(y_test,
y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', label="Training
score")
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', label="Cross-
validation score")
plt.title("Learning Curve of Gradient Boosting")
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()
```

Result of Model

In this learning curve, we can see that the training score is relatively high and stable, while the cross-validation score is much lower and fluctuates more. This indicates that the model is likely overfitting to the training data.

One possible reason for this is that the model has too many parameters and is able to fit the training data well, but is not able to generalize to unseen data as well. To fix this, we could try reducing the number of parameters in the model, for example by using dimensionality reduction techniques or by limiting the depth of the model.

Another possible reason could be that the model is not sufficiently

regularized, which means it is able to fit the training data well but is not able to generalize to unseen data. In this case, we could try adding regularization to the model, for example by increasing the regularization strength or by adding dropout layers. It is important to carefully tune and regularize the model to ensure that it is able to generalize well to unseen data, rather than just fitting the training data.

3.4.2.8 AdaBoostClassifier

In this code, I am using the AdaBoost classifier from the sklearn library to train a model on a dataset. I am using the `learning_curve` function to generate learning curves for the model, which show the relationship between the size of the training set and the model's performance.

Fitting the AdaBoost classifier to the training data and making predictions on the test data. I am evaluating the model's performance using several metrics, including log loss, mean cross-validation score, accuracy, and a confusion matrix. I am also generating a classification report, which provides detailed precision, recall, and f1-score values for each class. using the matplotlib library to plot the learning curves and visualize how the model's performance changes as the size of the training set increases.

This code is intended to evaluate the performance of the AdaBoost classifier on the given dataset and understand how well the model is able to learn from the training data. The results of these evaluations can be used to identify potential areas for improvement and fine-tune the model to achieve better performance.

```
from sklearn.ensemble import AdaBoostClassifier

train_sizes, train_scores, test_scores =
learning_curve(AdaBoostClassifier(), X_train_tfidf, y_train, cv=5)

ada = AdaBoostClassifier()
ada.fit(X_train_tfidf, y_train)
y_pred = ada.predict(X_test_tfidf)

loss = log_loss(y_test, y_pred)
scores = cross_val_score(ada, X_train_tfidf, y_train, cv=5)

print("Loss of AdaBoost: ", loss)
print("Mean cross validation score: ", scores.mean())
print("Accuracy of AdaBoost: ", accuracy_score(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
print(classification_report(y_test, y_pred))

plt.figure()
plt.plot(train_sizes, train_scores.mean(axis=1), 'o-', label="Training
score")
plt.plot(train_sizes, test_scores.mean(axis=1), 'o-', label="Cross-
validation score")
```

```
plt.title("AdaBoost Learning Curves")
plt.xlabel("Number of training examples")
plt.ylabel("Accuracy")
plt.legend(loc="best")
plt.show()
```

Result of Model

The learning curve for the AdaBoost model looks somewhat erratic. There is a large gap between the training score and the cross-validation score, which suggests that the model is overfitting on the training data. This can be caused by a number of factors, such as having a high number of features or a lack of regularization. To avoid or fix this, we can try decreasing the number of features or applying regularization techniques, such as adding a penalty term to the cost function or using techniques like early stopping or dropout. This can help to reduce overfitting and improve the generalization performance of the model.

4. USED METHODOLOGY

In my project, I am using a dataset that contains two columns: comments and comments_rates. The goal of this project is to perform sentiment analysis on the comments, and to use the comments_rates as a reference for evaluating the performance of the sentiment analysis model.

To achieve this, I have implemented a machine learning model using Python. I have chosen to use a machine learning model because it allows me to automatically learn patterns in the data and make predictions based on those patterns. I have selected and preprocessed the data, and split it into training and testing sets. I have also implemented a number of preprocessing steps, such as removing stop words and stemming, in order to prepare the data for analysis. Next, I have trained the model on the training data and evaluated its performance on the testing data. I have used a number of metrics, such as accuracy and F1 score, to evaluate the model's performance and to identify any areas for improvement.

One potential area for improvement is to try different machine learning algorithms or to tune the hyperparameters of the model to see if that leads to better performance. Another option is to try different preprocessing techniques or to add additional features to the data.

For this project, I followed the following methodology:

- **Data Preprocessing:** I started by collecting and preprocessing the data. This involved cleaning the text data, removing stopwords and punctuation, and stemming the words. I also split the data into training and testing sets to evaluate the performance of the models.

- **Feature Extraction:** I used the term frequency-inverse document frequency (TF-IDF) method to extract features from the text data. This helped to identify the most important words in each document and weight them according to their importance.
- **Model Selection:** I compared the performance of several different classification algorithms, including logistic regression, support vector machine, decision tree, random forest, and AdaBoost. I used a number of metrics, including accuracy, precision, recall, and F1 score, to evaluate the performance of each model.
- **Model Tuning:** I used grid search to fine-tune the parameters of the models to optimize their performance. This involved trying different combinations of parameters and selecting the ones that gave the best results.
- **Evaluation:** Finally, I evaluated the performance of the models using a number of metrics, including accuracy, precision, recall, and F1 score. I also plotted learning curves to visualize the training and cross-validation performance of each model.

This project was a good opportunity to apply my knowledge of machine learning and natural language processing to a real-world problem. I learned a lot about the different algorithms and techniques that can be used to classify text data, and I gained experience in selecting and tuning models to optimize their performance.

5. FINAL PREDICT DEMO

6. FUTURE WORKS

- One potential direction for future work in NLP sentiment analysis could be to expand the scope of the dataset and include a wider range of comments and rating labels. This could help to improve the generalizability of the models and provide a more comprehensive view of sentiment in the data.
- The development of models specifically designed to prevent bad commenting or hate speech using elastic search. Elastic search is a powerful search and analytics engine that allows users to easily search and analyze large volumes of text data. By incorporating elastic search into the sentiment analysis process, it may be possible to more efficiently and effectively identify and classify potentially problematic content.
- Using machine learning algorithms to analyze text data and identify specific words or phrases that are commonly associated with stress, depression, or potential threats. This could involve using techniques such as sentiment analysis or topic modeling to identify negative or aggressive language, or using word embeddings or other advanced NLP techniques to identify specific words or phrases that are commonly associated with stress, depression, or potential threats. Once these patterns or trends have been identified, the next step could be to develop algorithms or models that are specifically designed to identify and classify potentially criminal

or hateful language in real-time. This could involve using machine learning algorithms to analyze incoming text data in real-time and identify potentially problematic content, and then implementing measures to prevent or mitigate the spread of this content.

- Using elastic search to index and analyze large volumes of text data, and then use machine learning algorithms to identify patterns or trends that are indicative of bad commenting or hate speech. This could involve using techniques such as sentiment analysis or topic modeling to identify negative or aggressive language, or using word embeddings or other advanced NLP techniques to identify specific words or phrases that are commonly associated with bad commenting or hate speech.
- Finally, it could be useful to consider the context or source of the comments in the analysis, as this could provide additional insights into the sentiment expressed in the data. For example, analyzing comments from different countries or industries could reveal different patterns or trends in sentiment, and incorporating this contextual information into the models could further improve their accuracy and effectiveness.

As you can see, there are many potential avenues for future work in NLP sentiment analysis, and continued exploration of these areas could help to further advance the field and improve our ability to understand and analyze sentiment in text data.