Sentimental Analysis for Marketing

Introduction

Sentiment analysis, or opinion mining, is a vital tool for marketers. It involves analyzing online conversations, reviews, and social media content to understand public sentiment toward products, brands, or topics. This analysis helps marketers with:

- Customer insights
- Reputation management
- Competitive analysis
- Product development
- Targeted marketing campaigns
- Customer feedback
- Brand loyalty and advocacy
- Predictive analytics
- Customer segmentation
- Social listening

By extracting and interpreting sentiments, marketers can make data-driven decisions to improve products, manage reputation, and create effective marketing strategies.

Importing Libraries:

```
pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import seaborn as sns

from sklearn.model_selection import train import _test_split
```

- **import pandas as pd**: Imports the Pandas library, commonly used for data manipulation and analysis.
- **import numpy as np**: Imports the NumPy library, which is used for numerical and array operations.
- **import matplotlib.pyplot as plt**: Imports the Matplotlib library for data visualization, and 'plt' is used as an alias to make it easier to create plots.

- **import seaborn as sns**: Imports the Seaborn library, which is a data visualization library often used to create informative and attractive statistical graphics.
- **from sklearn.model_selection import train_test_split**: Imports the **train_test_split** function from Scikit-Learn, a machine learning library, which is used to split datasets into training and testing subsets for model development.

```
import warnings

warnings.filterwarnings('ignore')
```

Warnings Suppression:

- **import warnings**: Imports the 'warnings' module, which is used to manage warnings issued by Python or libraries.
- **warnings.filterwarnings('ignore')**: Sets the behavior to ignore (suppress) all warning messages. This is typically done to prevent warning messages from cluttering the output during data analysis and modeling.

Data Processing :

```
print(df['airline_sentiment'].value_counts())

sns.countplot(data=df,x='airline_sentiment')

plt.show()
```

print(df['airline_sentiment'].value_counts()) :

- **df['airline_sentiment']** extracts the "airline_sentiment" column from the DataFrame 'df.'
- **value_counts()** is a Pandas function that counts the occurrences of unique values in the selected column.
- **print()** is used to display the result of value_counts().

This line of code prints the counts of different sentiment values in the "airline_sentiment" column, giving you an overview of how many times each sentiment (e.g., positive, negative, neutral) appears in the dataset.

sns.countplot(data=df, x='airline_sentiment'):

- **sns.countplot()**: is a function from the Seaborn library for creating count plots, which are used to visualize the distribution of categorical data.
- **data=df**: specifies the DataFrame from which the data for plotting will be drawn.
- **x='airline_sentiment'** :specifies the column in the DataFrame that will be used as the x-axis variable in the count plot.

This line of code creates a count plot that visualizes the distribution of different sentiment categories in the "airline_sentiment" column. Each category (positive, negative, neutral) will be represented on the x-axis, and the count of each category will be displayed on the y-axis.

plt.show():

- **plt.show()** is used to display the plot created with Seaborn. It is necessary to actually show the plot on the screen.

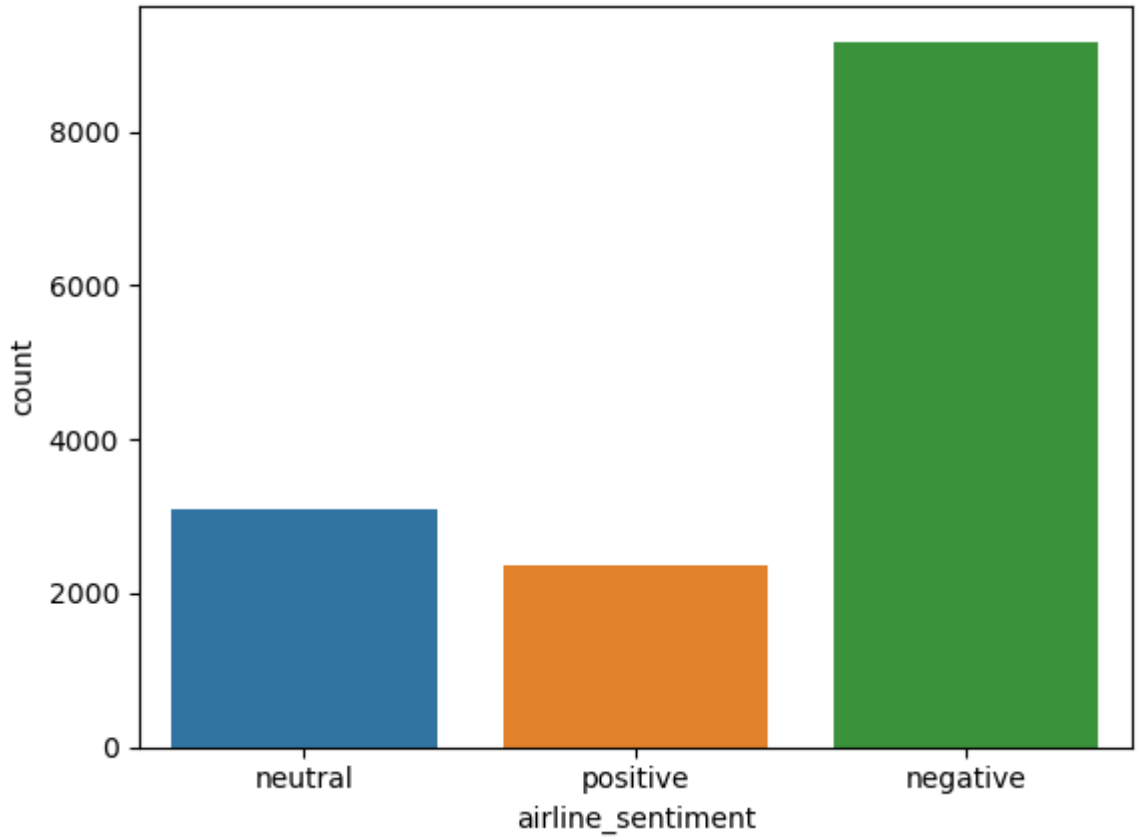
Output:

negative 9178

neutral 3099

positive 2363

Name: airline_sentiment, dtype: int64




```
df.airline.value_counts()
```

the code **df.airline.value_counts()** is a simple way to count how many times each unique airline appears in the "airline" column of the DataFrame 'df.' It provides a tally of how frequently each airline occurs in the dataset.

Output:

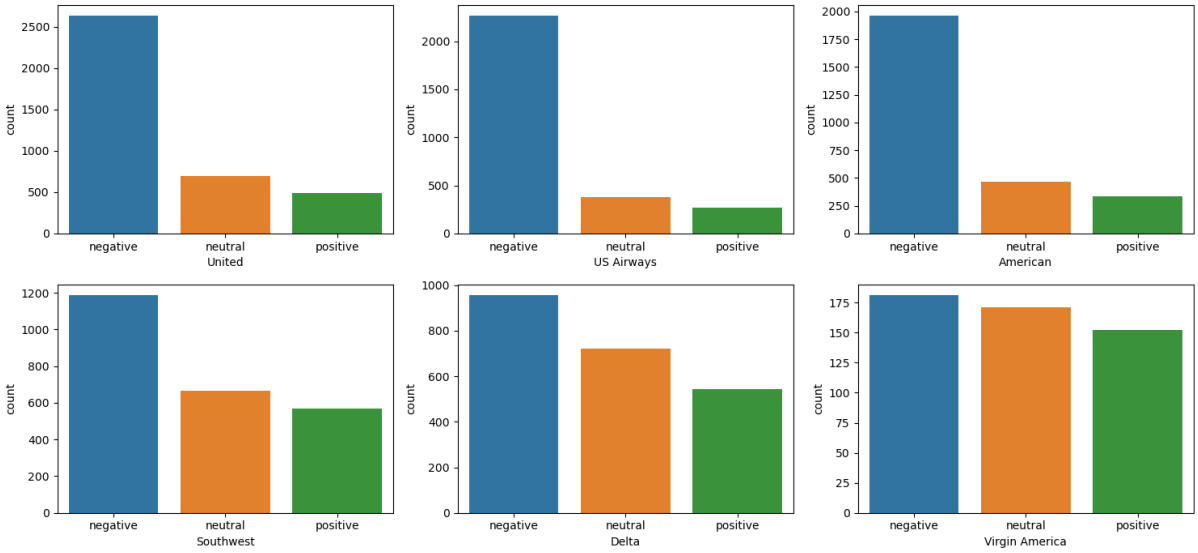
```
United          3822
US Airways      2913
American        2759
Southwest       2420
Delta           2222
Virgin America   504
Name: airline, dtype: int64
```

Facetted Data Visualization:

```
def plot_sentiment(Airline):
    df1 = df[df['airline']==Airline]

count=df1['airline_sentiment'].value_counts().reset_index().rename(columns={'index':Airline,'airline_sentiment':'count'})
    sns.barplot(data=count,x=Airline,y='count')
plt.figure(figsize=(15,7))
plt.subplot(2,3,1)
plot_sentiment('United')
plt.subplot(2,3,2)
plot_sentiment('US Airways')
plt.subplot(2,3,3)
plot_sentiment('American')
plt.subplot(2,3,4)
plot_sentiment('Southwest')
plt.subplot(2,3,5)
plot_sentiment('Delta')
plt.subplot(2,3,6)
plot_sentiment('Virgin America')
plt.tight_layout()
plt.show()
```

1. **def plot_sentiment(Airline)**: This is a Python function definition. It defines a function called **plot_sentiment** that takes one argument, **Airline**, which represents the name of an airline.
2. **df1 = df[df['airline'] == Airline]**: Within the **plot_sentiment** function, this line creates a new DataFrame **df1** by filtering the original DataFrame **df** to only include rows where the "airline" column matches the specified **Airline**. This effectively isolates the data for a specific airline.
3. **count = df1['airline_sentiment'].value_counts().reset_index().rename(columns={'index': Airline, 'airline_sentiment': 'count'})**: Within the **plot_sentiment** function, this line does the following:
    - **df1['airline_sentiment'].value_counts()**: It calculates the frequency of each unique sentiment value within the "airline_sentiment" column of the **df1** DataFrame. This provides a count of how many times each sentiment (e.g., positive, negative, neutral) appears for the specific airline.
    - **.reset_index()**: Resets the index of the resulting Series, making it a DataFrame with two columns: 'index' (containing the sentiment labels) and 'airline_sentiment' (containing the corresponding counts).
    - **.rename(columns={'index': Airline, 'airline_sentiment': 'count'})**: Renames the columns of the DataFrame to make it more descriptive. The 'index' column is renamed to the specific airline name (**Airline**), and the 'airline_sentiment' column is renamed to 'count'.
4. **plt.figure(figsize=(15, 7))**: This line creates a new Matplotlib figure with a specified size of 15 inches in width and 7 inches in height. This figure will contain the subplots (bar plots).
5. Subplots with **plt.subplot()**: The following lines create a grid of subplots (bar plots) using the **plt.subplot()** function. There are six subplots organized in a 2x3 grid.
    - **plt.subplot(2, 3, 1)**: This specifies the position of the first subplot in a 2x3 grid.
    - **plot_sentiment('United')**: Calls the **plot_sentiment** function with the airline name 'United' to create a bar plot for the sentiment distribution of that airline.
    - The subsequent lines follow the same pattern for different airlines ('US Airways,' 'American,' 'Southwest,' 'Delta,' and 'Virgin America').
6. **plt.tight_layout()**: This is used to ensure that the subplots are appropriately spaced and arranged for better visibility and readability.
7. **plt.show()**: Finally, this line displays the entire set of subplots in a single figure. Each subplot represents the sentiment distribution for a specific airline.

Output:



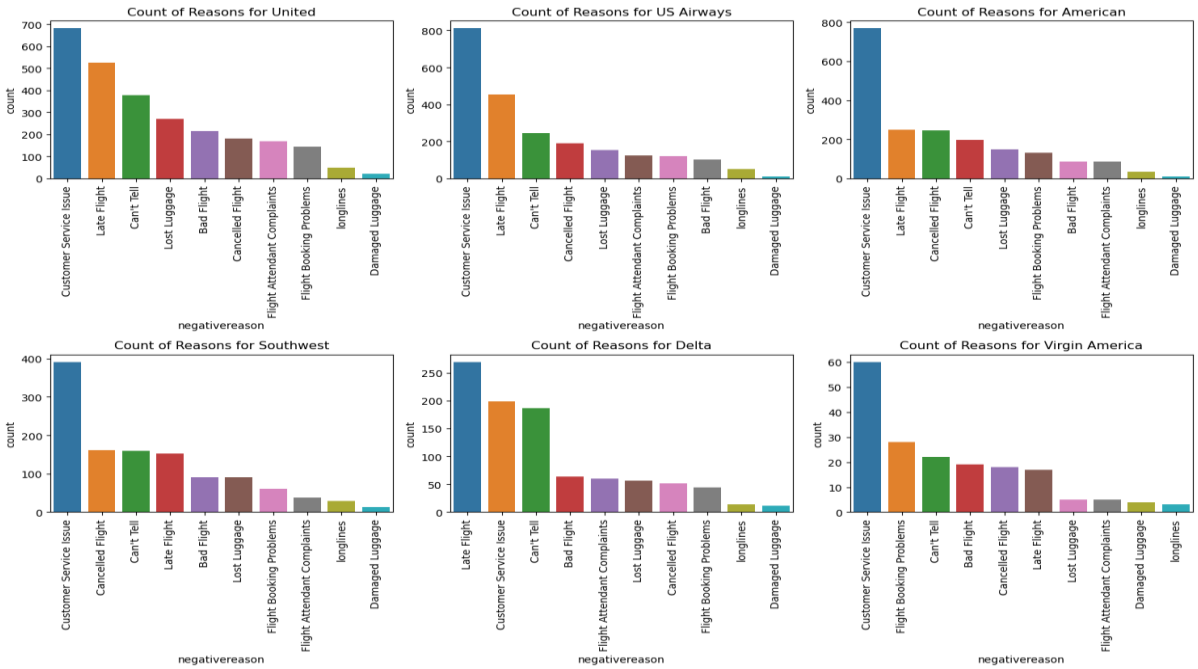

```
def plot_reason(Airline):
    df2 = df[df['airline']==Airline]

count=df2['negativereason'].value_counts().reset_index().rename(columns={'index':'negativereason','negativereason':'count'})
    sns.barplot(data=count,x='negativereason',y='count')
    plt.title('Count of Reasons for '+Airline)
    plt.xticks(rotation=90)
plt.figure(figsize=(15,10))
plt.subplot(2,3,1)
plot_reason('United')
plt.subplot(2,3,2)
plot_reason('US Airways')
plt.subplot(2,3,3)
plot_reason('American')
plt.subplot(2,3,4)
plot_reason('Southwest')
plt.subplot(2,3,5)
```

```
plot_reason('Delta')
plt.subplot(2,3,6)
plot_reason('Virgin America')
plt.tight_layout()
plt.show()
```

1. **def plot_reason(Airline)**: This is a function definition that takes an airline name, **Airline**, as its argument.
2. **df2 = df[df['airline']==Airline]**: Within the **plot_reason** function, this line creates a new DataFrame **df2** by filtering the original DataFrame 'df' to include only rows where the "airline" column matches the specified **Airline**. This isolates the data for a specific airline.
3. **count = df2['negativereason'].value_counts().reset_index().rename(columns={'index': 'negativereason', 'negativereason': 'count'})**:
    - It calculates the frequency of each unique reason for negative sentiment (stored in the "negativereason" column) within the **df2** DataFrame.
    - **.reset_index()** resets the index of the resulting Series, making it a DataFrame with two columns: 'negativereason' (containing the negative sentiment reasons) and 'count' (containing the corresponding counts).
    - **.rename(columns={'index': 'negativereason', 'negativereason': 'count'})** renames the columns for clarity.
4. **sns.barplot(data=count, x='negativereason', y='count')**: This line creates a bar plot using Seaborn. It visualizes the count of negative sentiment reasons for the specific airline. The x-axis represents the negative sentiment reasons, and the y-axis represents the count of each reason.
5. **plt.title('Count of Reasons for ' + Airline)**: Sets a title for the bar plot. The title indicates that the plot shows the count of reasons for negative sentiment for the specific airline.
6. **plt.xticks(rotation=90)**: Rotates the x-axis labels by 90 degrees to ensure they are readable when there are many categories (negative reasons) to display.
7. **plt.figure(figsize=(15, 10))**: Creates a new Matplotlib figure with a specified size of 15 inches in width and 10 inches in height. This figure will contain the subplots (bar plots).

8. Subplots with **plt.subplot()**: The following lines create a 2x3 grid of subplots, similar to the previous code. Each subplot displays the reasons for negative sentiment for a specific airline.
9. **plt.tight_layout()**: Ensures that the subplots are appropriately spaced and arranged for better visibility and readability.
10. **plt.show()**: Displays the entire set of subplots in a single figure. Each subplot represents the distribution of negative sentiment reasons for a specific airline.

Output:




```
from wordcloud import WordCloud,STOPWORDS

df3 = df[df['airline_sentiment']=='negative']

words = ' '.join(df3['text'])

cleaned_word = ' '.join([word for word in words.split() if 'http' not in word and not
word.startswith('@') and word!='RT'])

wordcloud = WordCloud(background_color='black',stopwords=STOPWORDS,

                width=3000, height=2500).generate(''.join(cleaned_word))

plt.imshow(wordcloud)

plt.axis('off')

plt.show()
```

1. **from wordcloud import WordCloud, STOPWORDS**: This line imports the **WordCloud** class and the **STOPWORDS** set from the **wordcloud** library. The **WordCloud** class is used to generate word cloud visualizations, and **STOPWORDS** contains a predefined set of common words that should be excluded from the word cloud.
2. **df3 = df[df['airline_sentiment']=='negative']**: This line creates a new DataFrame **df3** by filtering the original DataFrame **df**. It selects only those rows where the 'airline_sentiment' column has the value 'negative.' This assumes that the DataFrame **df** contains data related to airline sentiment, and this line is used to isolate negative sentiment data.
3. **words = ' '.join(df3['text'])**: This line extracts the 'text' column from the filtered DataFrame **df3** and joins all the text from this column into a single string, separated by spaces. This results in a long string containing all the text data from the negative sentiment tweets.
4. **cleaned_word = ' '.join([word for word in words.split() if 'http' not in word and not word.startswith('@') and word != 'RT'])**: Here, the **words** string is split into individual words, and a list comprehension is used to filter out specific words:
    - Words containing 'http' are excluded (to remove URLs).
    - Words starting with '@' are excluded (presumably Twitter usernames).
    - The word 'RT' is excluded, which is often used to denote retweets.

After filtering, the remaining words are joined into a single string, which will be used to generate the word cloud.

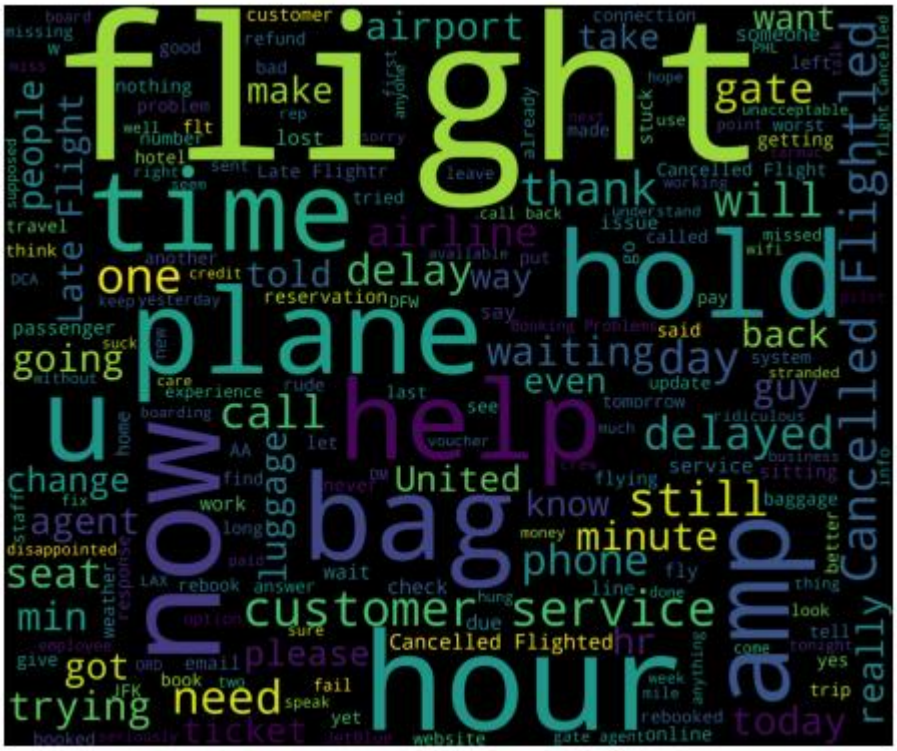
5. **wordcloud = WordCloud(background_color='black', stopwords=STOPWORDS, width=3000, height=2500).generate(''.join(cleaned_word))**: Here, a **WordCloud** object is created with the following parameters:
    - **background_color='black'**: This sets the background color of the word cloud to black.
    - **stopwords=STOPWORDS**: It specifies the set of stopwords to be excluded from the word cloud. This is based on the predefined **STOPWORDS** set imported earlier.
    - **width=3000** and **height=2500**: These parameters define the dimensions of the word cloud image.

The **generate** method is then called on the **WordCloud** object, and it takes the cleaned word string as input to generate the word cloud visualization.

6. **plt.imshow(wordcloud)**: This line displays the word cloud using the **imshow** function from the **matplotlib** library.
7. **plt.axis('off')**: It turns off the axis (i.e., the x and y axis labels) in the plot.

8. **plt.show()**: Finally, this command displays the word cloud visualization to the user.

Output:




```
from sklearn.linear_model import LogisticRegression

from sklearn.neighbors import KNeighborsClassifier

from sklearn.svm import SVC, LinearSVC, NuSVC

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier,
GradientBoostingClassifier

from sklearn.naive_bayes import GaussianNB

from sklearn.metrics import accuracy_score
```

1. Importing necessary modules:
    - **from sklearn.linear_model import LogisticRegression**: This line imports the **LogisticRegression** classifier from the scikit-learn (sklearn) library. Logistic regression is a common algorithm used for binary classification tasks.
    - **from sklearn.neighbors import KNeighborsClassifier**: This line imports the **KNeighborsClassifier** classifier, which is used for k-nearest neighbors classification.

- **from sklearn.svm import SVC, LinearSVC, NuSVC**: These lines import support vector machine (SVM) classifiers. **SVC** is the standard SVM classifier, **LinearSVC** is a linear SVM classifier, and **NuSVC** is an implementation of a nu-support vector classification.

2. Additional classifier imports:
    - **from sklearn.tree import DecisionTreeClassifier**: This line imports the **DecisionTreeClassifier**, which is used to create decision tree-based classification models.
    - **from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, GradientBoostingClassifier**: These lines import ensemble classifiers. **RandomForestClassifier** is an ensemble of decision trees (a random forest). **AdaBoostClassifier** is used for AdaBoost ensemble learning. **GradientBoostingClassifier** is an implementation of gradient boosting.
3. **from sklearn.naive_bayes import GaussianNB**: This line imports the **GaussianNB** classifier, which is a type of Naive Bayes classifier specifically designed for data with continuous features.
4. **from sklearn.metrics import accuracy_score**: This line imports the **accuracy_score** function from scikit-learn's **metrics** module. The **accuracy_score** function is used to calculate the accuracy of a classification model's predictions. It compares the predicted labels to the true labels and calculates the proportion of correctly classified samples.

```
dense_features=train_features.toarray()

dense_test= test_features.toarray()

Accuracy=[]

Model=[]

for classifier in Classifiers:

   try:

      fit = classifier.fit(train_features,train['sentiment'])

      pred = fit.predict(test_features)

except Exception:

      fit = classifier.fit(dense_features,train['sentiment'])

pred = fit.predict(dense_test)

accuracy = accuracy_score(pred,test['sentiment'])

Accuracy.append(accuracy)
```

```
Model.append(classifier.__class__.__name__)
print('Accuracy of '+classifier.__class__.__name__+'is '+str(accuracy))
result = pd.DataFrame({'Models':Model})
result['Accuracy'] = Accuracy
result = result.sort_values(by='Accuracy',ascending=False)
result
```

1. Converting Sparse Features to Dense Features:
    - **dense_features = train_features.toarray()**: This line converts the sparse training features represented in the **train_features** matrix into dense (or "dense array") format. Sparse features are typically stored in compressed format, and this conversion is necessary for certain classifiers that require dense input data.
    - **dense_test = test_features.toarray()**: Similarly, this line converts the sparse test features represented in the **test_features** matrix into a dense format.
2. Initialization of Lists for Storing Results:
    - **Accuracy = []**: This list, **Accuracy**, will be used to store the accuracy scores of each classifier on the test data.
    - **Model = []**: This list, **Model**, will be used to store the names of the classifier models used.
3. Iterating Over a List of Classifiers:
    - **for classifier in Classifiers:**: It appears that **Classifiers** is a list of machine learning classifiers (e.g., scikit-learn classifiers like LogisticRegression, RandomForestClassifier, etc.) that you want to evaluate.
4. Model Training and Prediction:
    - Inside the loop, the code tries to fit the classifier to the training data and make predictions on the test data:
        - **try:** and **except Exception:**: These are used for exception handling. The code tries to fit the classifier using **train_features** and make predictions on **test_features**. If an exception is raised (e.g., due to compatibility issues between the classifier and sparse data), it will catch the exception and instead fit the classifier using the dense feature representations (**dense_features**) and make predictions on **dense_test**.

- **fit = classifier.fit(...)**: This line fits the classifier to the training data. The specific method may vary depending on the classifier. It's essential for training the classifier on your data.
- **pred = fit.predict(...)**: This line uses the trained classifier to make predictions on the test data.

5. Calculating and Storing Accuracy:
    - **accuracy = accuracy_score(pred, test['sentiment'])**: This line calculates the accuracy of the classifier's predictions by comparing them to the true labels in the **test** dataset. It uses the **accuracy_score** function, which was imported earlier.
    - **Accuracy.append(accuracy)**: The accuracy score is appended to the **Accuracy** list to keep track of the performance of each classifier.
    - **Model.append(classifier.__class__.__name__)**: The name of the classifier (obtained using **__class__.__name__**) is appended to the **Model** list to keep track of which classifier corresponds to which accuracy score.
    - **print('Accuracy of ' + classifier.__class__.__name__ + ' is ' + str(accuracy))**: This line prints the accuracy of the current classifier.

Output:

Accuracy of LogisticRegressionis 0.6451502732240437

Accuracy of KNeighborsClassifieris 0.5580601092896175

Accuracy of SVCis 0.7435109289617486

Accuracy of DecisionTreeClassifieris 0.7609289617486339

Accuracy of RandomForestClassifieris 0.8183060109289617

Accuracy of AdaBoostClassifieris 0.7862021857923497

Accuracy of GaussianNBis 0.5846994535519126

	Models	Accuracy
4	RandomForestClassifier	0.818306
5	AdaBoostClassifier	0.786202

3	DecisionTreeClassifier	0.760929
2	SVC	0.743511
0	LogisticRegression	0.645150
6	GaussianNB	0.584699
1	KNeighborsClassifier	0.558060

Conclusion:

a sentiment analysis for marketing code is a powerful tool for understanding and leveraging customer feedback and sentiment to make data-driven marketing decisions. It provides insights into how customers perceive a product, service, or brand, and it helps identify areas for improvement and growth. The choice of machine learning models and their accuracy scores further guide marketing teams in selecting the most suitable approach for their specific marketing analytics needs.