SENTIMENT ANALYSIS OF MARKETING

Data analysis for sentiment analysis of marketing typically involves exploring and understanding the characteristics of your dataset to inform subsequent steps in your analysis. Below are steps for performing data analysis on a marketing dataset for sentiment analysis:

1. Load the Dataset:

- Import your marketing dataset into your chosen data analysis environment (e.g., Python with pandas).

```python
import pandas as pd

# Load your dataset
df = pd.read_csv("marketing_data.csv")
```

2. Data Overview:

- Start by getting a high-level understanding of your data by checking the first few rows and the basic statistics.

```python
# Display the first few rows
print(df.head())

# Get summary statistics for numerical columns
print(df.describe())
```

3. Data Cleaning:

- Check for missing values in the dataset and decide how to handle them (e.g., imputation or removal).

```python
# Check for missing values
print(df.isnull().sum())

# Handle missing values (if needed)
# df.dropna(subset=["text"], inplace=True)
```

4. **Data Distribution:**

- Analyze the distribution of sentiment labels in your dataset.

```python
sentiment_counts = df["Sentiment"].value_counts()
print(sentiment_counts)
```

5. **Text Preprocessing:**

- Preprocess the text data by techniques like lowercasing, tokenization, stop word removal, and lemmatization/stemming.

```python
# Example preprocessing (you can use NLTK, spaCy, or other libraries)
df["text"] = df["text"].str.lower()
```

```
# Tokenization, stop word removal, and lemmatization/stemming can also be applied here.
```

6. **Text Length Analysis**:

- Explore the distribution of text lengths, as it can affect sentiment analysis.

```python
df["Text Length"] = df["text"].apply(len)
print(df["Text Length"].describe())
```

7. **Word Frequency Analysis**:

- Analyze the most frequent words in both positive and negative marketing content to identify important keywords.

```python
# Example: Top words in positive and negative text
positive_texts = df[df["Sentiment"] == "Positive"]["text"]
negative_texts = df[df["Sentiment"] == "Negative"]["text"]

def get_top_words(texts, top_n=10):
    words = " ".join(texts).split()
    word_freq = pd.Series(words).value_counts()
    return word_freq.head(top_n)

print("Top words in positive content:")
print(get_top_words(positive_texts))
```

```
print("\nTop words in negative content:")
print(get_top_words(negative_texts))
```

8. **Visualization:**

- Create data visualizations to better understand the data, such as histograms, word clouds, or time series plots if your dataset includes timestamps.

```python
import matplotlib.pyplot as plt
import wordcloud

# Example: Word cloud for positive and negative texts
positive_wordcloud = wordcloud.WordCloud().generate(" ".join(positive_texts))
negative_wordcloud = wordcloud.WordCloud().generate(" ".join(negative_texts))

plt.figure(figsize=(10, 5))
plt.subplot(1, 2, 1)
plt.imshow(positive_wordcloud, interpolation="bilinear")
plt.title("Word Cloud for Positive Sentiments")

plt.subplot(1, 2, 2)
plt.imshow(negative_wordcloud, interpolation="bilinear")
plt.title("Word Cloud for Negative Sentiments")

plt.show()
```

9. **Correlation Analysis:**

- Examine potential correlations between numerical features and sentiment labels, if relevant.

```python
correlation_matrix = df.corr()
print(correlation_matrix)
```

10. **Additional Analyses**:

- Depending on your specific dataset, you may perform more advanced analyses, such as sentiment trends over time, topic modeling, or sentiment correlation with other marketing metrics.

Data analysis is a crucial step in the sentiment analysis process, helping you understand your data's characteristics, uncover potential biases or patterns, and guide your preprocessing and modeling decisions. The specific analyses you perform can vary depending on your dataset and research goals.

SENTIMENT ANALYSIS OF MARKETING

ABSTRACT:

```
import re

import sys

import numpy as np

import pandas as pd


import matplotlib.pyplot as plt

import seaborn as sns


import string

import nltk


import warnings

warnings.filterwarnings("ignore", category=DeprecationWarning)


from gensim.models import KeyedVectors

from sklearn.manifold import TSNE


%load_ext autoreload

%autoreload 2

%matplotlib inline
```

The above code sets up a Python environment for data analysis and natural language processing. It imports essential libraries, including NumPy, pandas, Matplotlib, Seaborn, and NLTK, for data manipulation and visualization. It also suppresses DeprecationWarnings, ensuring a clean output. The code loads the Gensim library for word embeddings and scikit-learn's t-SNE for dimensionality reduction. Additionally, it configures IPython-specific settings for module autoreloading and inline Matplotlib plots when used in Jupyter Notebook or IPython. This code snippet serves as a foundational step to create a robust

environment for various data analysis and NLP tasks.

```
train = pd.read_csv('../input/twitter-tweets-data/train_tweet.csv')

test = pd.read_csv('../input/twitter-tweets-data/test_tweets.csv')
```

```
print(train.shape)

print(test.shape)
```

This code reads and loads data from two CSV files, 'train_tweet.csv' and 'test_tweets.csv', into two separate Pandas DataFrames, 'train' and 'test'. The '../input/twitter-tweets-data/' part of the file paths suggests that these CSV files are located in a directory with that relative path. After successfully loading the data, the code prints the shapes of both DataFrames using the 'shape' attribute. The 'train' DataFrame represents the training data, while the 'test' DataFrame represents the testing data. Printing their shapes provides an initial understanding of the data's dimensions, showing the number of rows and columns in each DataFrame. This step is crucial for getting an overview of the dataset's size and structure, which is often the first step in any data analysis or machine learning project to ensure that the data was loaded correctly and to determine the data's basic characteristics.

The code train.head() is used to display the first few rows of the 'train' DataFrame. When you execute this code, it will print a table-like structure to the console, showing the top rows of the DataFrame. This is a common practice in data analysis to quickly inspect the data and get a sense of its content and structure.

```
train.isnull().any()

test.isnull().any()
```

The code `train.isnull().any()` checks the 'train' DataFrame for missing values and returns a Boolean Series that indicates whether there are any missing values (True) or not (False) for each column. Similarly, `test.isnull().any()` performs the same check for the 'test' DataFrame. This is a quick way to identify columns with missing data. If a column has at least one missing value, the corresponding entry in the Boolean Series will be True. It's a fundamental step in data preprocessing to address missing values, as they can affect the quality of analysis and modeling.

```
train.groupby('len').mean()['label'].plot.hist(color = 'black', figsize = (6, 4),)

plt.title('variation of length')

plt.xlabel('Length')

plt.show()
```

The code conducts an analysis of text length and sentiment labels in the 'train' DataFrame using a bar plot. It groups the data by text length ('len') and calculates the mean of the 'label' column within each length group. The 'label' column typically represents sentiment, such as positive or negative. The plot is displayed in black, with a specified figure size. The title 'Variation of Length' and labels for the x-axis ('Length') are added for clarity. The bar plot provides insights into how text length relates to sentiment. By examining the plot, you can identify patterns or trends in sentiment with respect to the length of the text.

```
from sklearn.feature_extraction.text import CountVectorizer



cv = CountVectorizer(stop_words = 'english')

words = cv.fit_transform(train.tweet)



sum_words = words.sum(axis=0)



words_freq = [(word, sum_words[0, i]) for word, i in cv.vocabulary_.items()]

words_freq = sorted(words_freq, key = lambda x: x[1], reverse = True)



frequency = pd.DataFrame(words_freq, columns=['word', 'freq'])



frequency.head(30).plot(x='word', y='freq', kind='bar', figsize=(15, 7), color = 'blue')

plt.title("Most Frequently Occuring Words - Top 30")
```

This code uses scikit-learn's CountVectorizer to analyze text data from the 'train' DataFrame:

1. It imports the necessary library, scikit-learn's CountVectorizer, for text analysis.

2. A CountVectorizer object, `cv`, is initialized to convert text data into numerical form, removing common English stop words to improve accuracy.

3. The 'train.tweet' column is transformed into a word count matrix using `cv.fit_transform(train.tweet)`.

4. The sum of word counts across all documents is calculated and stored in 'sum_words'.

5. A list of word-frequency pairs is created, sorted by frequency in descending order.

6. A Pandas DataFrame named 'frequency' is constructed from the sorted word-frequency pairs with columns 'word' and 'freq'.

7. A bar plot is generated to visualize the top 30 most frequently occurring words, with words on the x-axis and their frequencies on the y-axis.

8. The plot is displayed in blue, and its figure size is adjusted for clarity.

9. The plot is given the title "Most Frequently Occurring Words - Top 30".

10. This analysis helps identify common words in the text data, which is valuable for tasks like text preprocessing and understanding the dataset's characteristics.

```
from wordcloud import WordCloud

wordcloud = WordCloud(background_color = 'white', width = 1000, height =
1000).generate_from_frequencies(dict(words_freq))

plt.figure(figsize=(10,8))

plt.imshow(wordcloud)

plt.title("WordCloud - Vocabulary from Reviews", fontsize = 22)
```

This code utilizes the 'WordCloud' library to generate a word cloud visualization, which offers a concise and visually striking representation of the most frequently occurring words in a text dataset. The word cloud is configured with a white background and specific dimensions (width and height), and it is created from a dictionary containing word-frequency pairs, 'dict(words_freq),' where words with higher frequencies will appear more prominently. The 'plt.figure' command sets up the figure size for the visualization, 'plt.imshow' displays the actual word cloud, and a title, "WordCloud - Vocabulary from Reviews," is added to provide context. This type of visualization is commonly used in text analysis to quickly identify and understand the most common terms within a corpus, making it a valuable tool for extracting key themes or topics from text data. The larger and bolder words in the word cloud represent the most prevalent terms, offering a straightforward way to gain insights into the dataset's content.

```
normal_words =' '.join([text for text in train['tweet'][train['label'] == 0]])


wordcloud = WordCloud(width=800, height=500, random_state = 0, max_font_size = 110).generate(normal_words)

plt.figure(figsize=(10, 7))

plt.imshow(wordcloud, interpolation="bilinear")

plt.axis('off')

plt.title('The Neutral Words')

plt.show()
```

This code creates a word cloud visualization to display the most frequent words found in the text data labeled as "neutral" in a dataset. It starts by extracting all the text associated with neutral sentiment (label = 0) from the 'train' DataFrame and joins them into a single string variable 'normal_words.' Next, the WordCloud library is employed to generate the word cloud. Specific parameters are set, such as the width, height, a random state for reproducibility, and the maximum font size for the words. A figure with a defined size is prepared, and the word cloud is displayed using 'plt.imshow.' By specifying 'interpolation="bilinear",' the word cloud's appearance is enhanced for readability. The 'plt.axis("off")' command removes axis labels and ticks. Finally, the title "The Neutral Words" is added to provide context, and the word cloud visualization is presented. This visual representation helps identify and highlight the most common words associated with neutral sentiment, which can offer valuable insights into the characteristics of neutral text in the dataset.

```
negative_words =' '.join([text for text in train['tweet'][train['label'] == 1]])
```

```
wordcloud = WordCloud(background_color = 'red', width=800, height=500, random_state = 0, max_font_size = 110).generate(negative_words)

plt.figure(figsize=(10, 7))

plt.imshow(wordcloud, interpolation="bilinear")

plt.axis('off')

plt.title('The Negative Words')

plt.show()
```

This code is used to create a word cloud visualization of the most frequently occurring words within text data labeled as "negative" (label = 1) in a dataset. It starts by extracting all the text associated with negative sentiment from the 'train' DataFrame and concatenates them into a single string variable named 'negative_words.' Next, the WordCloud library is employed to generate the word cloud with specific settings, such as a red background, width, height, random state for reproducibility, and the maximum font size for the words. A Matplotlib figure is prepared with a specified size, and the word cloud is displayed using 'plt.imshow.' The 'interpolation="bilinear"' setting enhances the appearance of the word cloud for readability. The 'plt.axis("off")' command removes axis labels and ticks. Lastly, the title "The Negative Words" is added to provide context, and the word cloud visualization is presented. This visualization helps to identify and emphasize the most common words associated with negative sentiment in the dataset, offering valuable insights into the characteristics of negative text data.

```
# collecting the hashtags


def hashtag_extract(x):

    hashtags = []


    for i in x:

        ht = re.findall(r"#(\w+)", i)

        hashtags.append(ht)
```

```
    return hashtags
```

The provided code defines a Python function, `hashtag_extract`, designed to extract hashtags from a list of text strings. It accomplishes this by iterating through each text string in the input list, using a regular expression to find substrings that start with "#" and are followed by one or more word characters. The hashtags found in each text string are stored in an inner list, and all these lists are collected in the `hashtags` list. This function is particularly handy when working with social media data or any text data that includes hashtags. By extracting hashtags, it allows for analysis of popular topics or themes within the text, which can be valuable in understanding trends or sentiment associated with specific hashtags. The resulting list of lists provides a structured way to access the extracted hashtags for each text string.

```
# extracting hashtags from non racist/sexist tweets

HT_regular = hashtag_extract(train['tweet'][train['label'] == 0])


# extracting hashtags from racist/sexist tweets

HT_negative = hashtag_extract(train['tweet'][train['label'] == 1])


# unnesting list

HT_regular = sum(HT_regular,[])

HT_negative = sum(HT_negative,[])
```

This code is designed to extract and organize hashtags from two distinct categories of tweets within a dataset: non-racist/sexist tweets and racist/sexist tweets. It starts by applying the `hashtag_extract` function to the text of non-racist/sexist tweets (label = 0) and stores the extracted hashtags in the `HT_regular` list. Similarly, it extracts hashtags from the text of racist/sexist tweets (label = 1) and stores them in the `HT_negative` list. To make the data more amenable for analysis, the code then "unnests" these lists, effectively flattening them into single lists of hashtags for each category. This process is beneficial when you want to examine and visualize the hashtags separately for each tweet category. By isolating and organizing the hashtags in this way, researchers and data analysts can gain insights into the trending topics or themes within each category of tweets, helping to understand the sentiment and trends associated with non-racist/sexist and racist/sexist content in the dataset.

```
a = nltk.FreqDist(HT_regular)

d = pd.DataFrame({'Hashtag': list(a.keys()),
```

```
                'Count': list(a.values())})


# selecting top 20 most frequent hashtags

d = d.nlargest(columns="Count", n = 20)

plt.figure(figsize=(16,5))

ax = sns.barplot(data=d, x= "Hashtag", y = "Count")

ax.set(ylabel = 'Count')

plt.show()
```

This code conducts an analysis of the most frequently occurring hashtags within a dataset of non-racist/sexist tweets. It begins by using the Natural Language Toolkit (NLTK) to create a frequency distribution, `nltk.FreqDist`, for the hashtags found in non-racist/sexist tweets (`HT_regular`). This distribution captures the count of each unique hashtag. The code then constructs a Pandas DataFrame, 'd,' with two columns: 'Hashtag' and 'Count,' where 'Hashtag' contains the unique hashtags and 'Count' corresponds to their frequency.

Subsequently, the code selects the top 20 most frequent hashtags by sorting the DataFrame 'd' based on the 'Count' column in descending order using `d.nlargest()`.

Finally, the code generates a horizontal bar plot using Seaborn, visualizing the selected top 20 hashtags. The hashtags are shown on the x-axis, their respective counts on the y-axis, and the plot is displayed in a larger format for clarity. This visualization allows analysts to quickly grasp which hashtags are most prevalent in non-racist/sexist tweets, providing insights into the popular themes or topics within this category of content. It is a valuable tool for understanding the characteristics of non-racist/sexist content and identifying trends within the dataset.

```
# tokenizing the words present in the training set

tokenized_tweet = train['tweet'].apply(lambda x: x.split())


# importing gensim

import gensim
```

```
# creating a word to vector model

model_w2v = gensim.models.Word2Vec(
        tokenized_tweet,
        size=200, # desired no. of features/independent variables
        window=5, # context window size
        min_count=2,
        sg = 1, # 1 for skip-gram model
        hs = 0,
        negative = 10, # for negative sampling
        workers= 2, # no.of cores
        seed = 34)


model_w2v.train(tokenized_tweet, total_examples= len(train['tweet']), epochs=20)
```

This code involves word tokenization, word vectorization, and model training using the Gensim library. First, it tokenizes the words in the training set by splitting each tweet into a list of words. Next, it imports Gensim, a natural language processing library. It then creates a Word2Vec model, 'model_w2v,' for word embedding. This model is configured with various parameters, specifying the desired number of features, context window size, minimum word frequency, and training specifics like skip-gram model, negative sampling, and multi-core processing. Finally, it trains the Word2Vec model using the tokenized tweets from the training data, providing a word embedding model that captures semantic relationships among words. This process is crucial for representing words as dense vectors, enabling various natural language processing tasks such as word similarity, document classification, and more.

```
model_w2v.wv.most_similar(positive = "dinner")
```

This line of code uses the trained Word2Vec model to find words most similar to "dinner" in terms of their word embeddings.

```
model_w2v.wv.most_similar(positive = "cancer")
```

This line of code utilizes the Word2Vec model to identify words most similar to "cancer" based on their word embeddings.

```
model_w2v.wv.most_similar(positive = "apple")
```

This code employs the Word2Vec model to identify words most similar to "apple" based on their word embeddings.

```
model_w2v.wv.most_similar(negative = "hate"
```

This code uses the Word2Vec model to find words that are dissimilar to "hate" based on their word embeddings.

```
from tqdm import tqdm

tqdm.pandas(desc="progress-bar")

from gensim.models.doc2vec import LabeledSentence
```

This code snippet involves the use of the `tqdm` library to create a progress bar for monitoring the progress of various tasks in the code. The `tqdm.pandas(desc="progress-bar")` line configures `tqdm` to display a progress bar with a description of "progress-bar" during the execution of tasks, making it easier to track the progress of time-consuming operations. Additionally, the code imports the `LabeledSentence` class from Gensim's `doc2vec` module, which is commonly used for document embedding tasks like creating paragraph vectors or document similarity tasks. This class helps in structuring labeled sentences or documents for subsequent modeling. Overall, this code snippet enhances the code's user-friendliness by providing a progress bar and introduces a key component for document embedding using Gensim's Doc2Vec model.

```
def add_label(twt):
  output = []
  for i, s in zip(twt.index, twt):
    output.append(LabeledSentence(s, ["tweet_" + str(i)]))
  return output

# label all the tweets
labeled_tweets = add_label(tokenized_tweet)

labeled_tweets[:6]
```

The code defines a function, `add_label(twt)`, which takes a list of tokenized tweets (`twt`) as input. It processes each tweet by iterating through the list and assigning a unique label to each tweet using the `LabeledSentence` class from Gensim. The label consists of "tweet_"

followed by the index of the tweet within the list. The function collects these labeled tweets into a new list, 'output,' and returns it.

After defining the function, it's used to label all the tokenized tweets stored in 'tokenized_tweet.' The result is a list called 'labeled_tweets' containing the tokenized tweets with unique labels. This labeling is a crucial step when preparing data for Doc2Vec or similar models, as it associates each text with a unique identifier, facilitating subsequent analysis and model training. The code provides a preview of the first six labeled tweets to illustrate the format of these labeled data points.

```
# removing unwanted patterns from the data


import re

import nltk


nltk.download('stopwords')

from nltk.corpus import stopwords

from nltk.stem.porter import PorterStemmer
```

This code serves to preprocess and clean text data by removing unwanted patterns and preparing it for analysis. It begins by importing the 're' module for regular expressions and the 'nltk' library for natural language processing. The code also downloads the stopwords corpus from NLTK, which includes common words (e.g., "the," "is") that are often removed to reduce noise in text data. Lastly, it imports the Porter Stemmer from NLTK, a tool for stemming words to their base form, which helps in standardizing and simplifying text for analysis. These preparations are fundamental to text processing and analysis, allowing for the removal of unnecessary elements and text normalization.

```
train_corpus = []


for i in range(0, 31962):

  review = re.sub('[^a-zA-Z]', ' ', train['tweet'][i])

  review = review.lower()

  review = review.split()
```

```
ps = PorterStemmer()


# stemming

review = [ps.stem(word) for word in review if not word in set(stopwords.words('english'))]


# joining them back with space

review = ' '.join(review)

train_corpus.append(review)
```

This code segment preprocesses text data from the 'train' dataset for further analysis, specifically focusing on cleaning and preparing the tweets for natural language processing tasks:

1. `train_corpus = []`: It initializes an empty list named 'train_corpus' to store the processed and cleaned text data.

2. The following loop iterates through each row in the 'train' dataset (a total of 31,962 rows):

   a. `review = re.sub('[^a-zA-Z]', ' ', train['tweet'][i])`: It uses a regular expression to remove any characters that are not alphabetic letters from the current tweet ('tweet[i]'). This step cleans the text by retaining only alphabetic characters.

   b. `review = review.lower()`: It converts all text to lowercase to ensure consistency in text analysis and reduce the impact of letter case.

   c. `review = review.split()`: It splits the text into a list of words.

   d. `ps = PorterStemmer()`: An instance of the Porter Stemmer from the NLTK library is created for word stemming.

e. `review = [ps.stem(word) for word in review if not word in set(stopwords.words('english'))]`: It stems each word in the list and removes common English stopwords, as defined by NLTK's stopwords list. This helps in standardizing and simplifying the text for analysis while eliminating common noise words.

f. `review = ' '.join(review)`: It joins the cleaned and stemmed words back together into a single text string.

g. `train_corpus.append(review)`: The processed text is added to the 'train_corpus' list.

The end result is a 'train_corpus' list containing cleaned and preprocessed text data, which is often used for various natural language processing tasks such as sentiment analysis or text classification.

```
test_corpus = []

for i in range(0, 17197):
  review = re.sub('[^a-zA-Z]', ' ', test['tweet'][i])
  review = review.lower()
  review = review.split()

  ps = PorterStemmer()

  # stemming
  review = [ps.stem(word) for word in review if not word in set(stopwords.words('english'))]

  # joining them back with space
  review = ' '.join(review)
  test_corpus.append(review)
```

This code segment is responsible for cleaning and preparing text data in the 'test' dataset for natural language processing tasks. It follows these key steps:

1. `test_corpus = []`: It initializes an empty list to store cleaned text data.

2. A loop processes each tweet in the 'test' dataset, one by one.

3. Text cleaning includes converting text to lowercase, removing non-alphabetic characters, splitting into words, stemming words, and removing common English stopwords.

4. The cleaned words are then rejoined into a single text string.

5. The processed text is appended to the 'test_corpus' list, providing standardized and prepared text data for analysis and modeling.

```
# creating bag of words


from sklearn.feature_extraction.text import CountVectorizer


cv = CountVectorizer(max_features = 2500)

x = cv.fit_transform(train_corpus).toarray()

y = train.iloc[:, 1]


print(x.shape)

print(y.shape)
```

This code segment is focused on creating a bag of words representation for the 'train_corpus' text data. It begins by importing the CountVectorizer from scikit-learn, which is a tool for converting text data into numerical vectors. A CountVectorizer is configured with a maximum of 2500 features (vocabulary size). It then transforms the 'train_corpus' text data into a numerical format with word counts. The resulting 'x' array contains the bag of words representation. 'y' is assigned to the labels from the 'train' dataset. Finally, the code prints the shapes of 'x' and 'y,' providing insights into the dimensions of the bag of words matrix and the label vector, respectively. This preprocessing step is crucial for text classification and machine learning tasks, as it converts text data into a format suitable for modeling and analysis.

```
# creating bag of words


from sklearn.feature_extraction.text import CountVectorizer
```

```
cv = CountVectorizer(max_features = 2500)

x_test = cv.fit_transform(test_corpus).toarray()


print(x_test.shape)
```

This code segment is responsible for creating a bag of words representation for the 'test_corpus' text data. It starts by importing the CountVectorizer from scikit-learn, a tool for converting text data into numerical vectors. The CountVectorizer is configured with a maximum of 2500 features (vocabulary size), and then it transforms the 'test_corpus' text data into a numerical format with word counts. The resulting 'x_test' array contains the bag of words representation for the test data. Finally, the code prints the shape of 'x_test,' providing information about the dimensions of the bag of words matrix for the test data. This preprocessing step is essential for text classification and machine learning tasks, as it converts text data into a format suitable for analysis and model predictions.

```
# splitting the training data into train and valid sets


from sklearn.model_selection import train_test_split


x_train, x_valid, y_train, y_valid = train_test_split(x, y, test_size = 0.25, random_state = 42)


print(x_train.shape)

print(x_valid.shape)

print(y_train.shape)

print(y_valid.shape)
```

This code segment is responsible for splitting the training data into training and validation sets, a common practice in machine learning. It uses scikit-learn's `train_test_split` function to divide the features ('x') and labels ('y') into two subsets, typically for model training and evaluation. The 'test_size' parameter is set to 0.25, meaning that 25% of the data will be allocated to the validation set, while the remaining 75% becomes the training set. The 'random_state' parameter ensures reproducibility of the split. Finally, the code prints the shapes of the training and validation data, providing insights into the dimensions of these two subsets, which are essential for model development and assessment.

```
# standardization
```

```
from sklearn.preprocessing import StandardScaler


sc = StandardScaler()


x_train = sc.fit_transform(x_train)

x_valid = sc.transform(x_valid)

x_test = sc.transform(x_test)
```

This code segment is responsible for standardizing the feature data, which is a common preprocessing step in machine learning. It employs scikit-learn's `StandardScaler` to ensure that the data has a mean of 0 and a standard deviation of 1, making it more suitable for various machine learning algorithms. It separately scales the training, validation, and test data, ensuring that the scaling parameters are based solely on the training data to prevent data leakage. Standardization helps in improving the convergence and performance of many machine learning models, particularly those that rely on the magnitude of features, such as support vector machines or k-nearest neighbors.

```
from sklearn.ensemble import RandomForestClassifier

from sklearn.metrics import confusion_matrix

from sklearn.metrics import f1_score


model = RandomForestClassifier()

model.fit(x_train, y_train)


y_pred = model.predict(x_valid)


print("Training Accuracy :", model.score(x_train, y_train))

print("Validation Accuracy :", model.score(x_valid, y_valid))


# calculating the f1 score for the validation set

print("F1 score :", f1_score(y_valid, y_pred))
```

```
# confusion matrix

cm = confusion_matrix(y_valid, y_pred)

print(cm)
```

This code uses the scikit-learn library to implement a Random Forest Classifier for a machine learning task. It begins by importing the necessary modules, including RandomForestClassifier for model creation and evaluation metrics such as confusion_matrix and f1_score. The Random Forest model is trained on the training data ('x_train' and 'y_train') using the 'fit' method. Predictions are then made on the validation set ('x_valid'), and the training and validation accuracies are calculated and printed. Additionally, the code computes and displays the F1 score, a metric that combines precision and recall, offering insights into the model's overall performance. Finally, a confusion matrix is generated to visualize the model's classification results, helping to understand how well it predicts different classes. This code showcases the complete process of training, evaluating, and diagnosing a machine learning model using a Random Forest classifier.

```
from sklearn.linear_model import LogisticRegression


model = LogisticRegression()

model.fit(x_train, y_train)


y_pred = model.predict(x_valid)


print("Training Accuracy :", model.score(x_train, y_train))

print("Validation Accuracy :", model.score(x_valid, y_valid))


# calculating the f1 score for the validation set

print("f1 score :", f1_score(y_valid, y_pred))


# confusion matrix

cm = confusion_matrix(y_valid, y_pred)

print(cm)
```

This code employs scikit-learn's Logistic Regression model for a machine learning task. It begins by importing the necessary modules, including LogisticRegression for model creation and evaluation metrics like confusion_matrix and f1_score. The Logistic Regression model is trained on the training data ('x_train' and 'y_train') using the 'fit' method. Predictions are made on the validation set ('x_valid'), and both training and validation accuracies are computed and displayed. Furthermore, the code calculates and reports the F1 score, a metric that combines precision and recall to assess the overall model performance. Finally, a confusion matrix is generated to visualize the model's classification results, aiding in understanding how effectively it predicts different classes. This code demonstrates the complete process of training, assessing, and diagnosing a machine learning model using a Logistic Regression classifier.

```
from sklearn.tree import DecisionTreeClassifier



model = DecisionTreeClassifier()

model.fit(x_train, y_train)



y_pred = model.predict(x_valid)



print("Training Accuracy :", model.score(x_train, y_train))

print("Validation Accuracy :", model.score(x_valid, y_valid))



# calculating the f1 score for the validation set

print("f1 score :", f1_score(y_valid, y_pred))



# confusion matrix

cm = confusion_matrix(y_valid, y_pred)

print(cm)
```

This code snippet employs scikit-learn's Decision Tree Classifier for a machine learning task. It initiates the model by importing the necessary modules, including DecisionTreeClassifier for model creation and evaluation metrics like confusion_matrix and f1_score. The Decision Tree model is trained on the training data ('x_train' and 'y_train') using the 'fit' method. Subsequently, predictions are generated on the validation set ('x_valid'), and both training and validation accuracies are computed and displayed. Additionally, the code computes and reports the F1 score, a metric that combines precision and recall to evaluate the model's

overall performance. Finally, a confusion matrix is generated to provide a visual representation of the model's classification results, aiding in understanding its effectiveness in predicting different classes. This code demonstrates the complete process of training, evaluating, and analyzing a machine learning model using a Decision Tree classifier.

```
from sklearn.svm import SVC


model = SVC()

model.fit(x_train, y_train)


y_pred = model.predict(x_valid)


print("Training Accuracy :", model.score(x_train, y_train))

print("Validation Accuracy :", model.score(x_valid, y_valid))


# calculating the f1 score for the validation set

print("f1 score :", f1_score(y_valid, y_pred))


# confusion matrix

cm = confusion_matrix(y_valid, y_pred)

print(cm)
```

This code segment utilizes scikit-learn's Support Vector Classifier (SVC) for a machine learning task. It starts by importing the necessary modules, including SVC for model creation and evaluation metrics like confusion_matrix and f1_score. The SVC model is trained on the training data ('x_train' and 'y_train') using the 'fit' method. Predictions are made on the validation set ('x_valid'), and both training and validation accuracies are calculated and displayed. Additionally, the code calculates and reports the F1 score, a metric that combines precision and recall to evaluate the model's overall performance. Finally, a confusion matrix is generated to visually represent the model's classification results, assisting in understanding its effectiveness in predicting different classes. This code illustrates the complete process of training, assessing, and analyzing a machine learning model using a Support Vector Classifier (SVC).

```
from xgboost import XGBClassifier
```

```
model = XGBClassifier()

model.fit(x_train, y_train)



y_pred = model.predict(x_valid)



print("Training Accuracy :", model.score(x_train, y_train))

print("Validation Accuracy :", model.score(x_valid, y_valid))



# calculating the f1 score for the validation set

print("f1 score :", f1_score(y_valid, y_pred))



# confusion matrix

cm = confusion_matrix(y_valid, y_pred)

print(cm)
```

This code snippet employs the XGBoost (Extreme Gradient Boosting) classifier for a machine learning task. It starts by importing the necessary modules, including XGBClassifier for model creation and evaluation metrics like confusion_matrix and f1_score. The XGBoost model is trained on the training data ('x_train' and 'y_train') using the 'fit' method. Predictions are generated on the validation set ('x_valid'), and both training and validation accuracies are computed and displayed. Additionally, the code calculates and reports the F1 score, a metric combining precision and recall to assess the model's overall performance. Finally, a confusion matrix is generated to visually represent the model's classification results, helping understand its effectiveness in predicting different classes. This code demonstrates the entire process of training, evaluating, and analyzing a machine learning model using the XGBoost classifier.