
**Name –Aditya Panditrao**

**Reg No -22MCB0032**

**College Name - Vellore Institute of Technology**

**Email id – adityapanditrao1612@gmail.com**

# Task 6. Data Science example.

Perform a Text Classification on consumer complaint dataset (https://catalog.data.gov/dataset/consumer-complaint-database) into following categories.

| | |
|---|---|
| 0 | Credit reporting, repair, or other |
| 1 | Debt collection |
| 2 | Consumer Loan |
| 3 | Mortgage |

Steps to be followed -

1. Explanatory Data Analysis and Feature Engineering
2. Text Pre-Processing
3. Selection of Multi Classification model
4. Comparison of model performance
5. Model Evaluation
6. Prediction

## Solution for task 6 –

**"The following steps show the code along with its explanations."**

**Step -1**

```
!wget https://files.consumerfinance.gov/ccdb/complaints.csv.zip
```

**Explanation-**

1.**! wget**: This part of the code invokes the **wget** command. The exclamation mark (**!**) is often used in Jupyter Notebook or Jupyter Lab environments to run shell commands from within a notebook cell.

**2. https://files.consumerfinance.gov/ccdb/complaints.csv.zip**: This is the URL from which the file will be downloaded. It points to a ZIP-compressed CSV file named "complaints.csv.zip" hosted on the Consumer Financial Protection Bureau (CFPB) website. This file likely contains consumer complaint data.

**Step -2**

```
!unzip complaints.csv.zip

Archive:  complaints.csv.zip
  inflating: complaints.csv
```

**Explanation-**

- unzip: This is the actual command being executed. It's a standard command-line utility for extracting files from ZIP archives.
- complaints.csv.zip: This is the name of the ZIP file that you want to extract.

**Step -3**

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter
```

**Explanation-**

1. **import pandas as pd**: This line imports the Pandas library and aliases it as "pd." Pandas is a popular Python library for data manipulation and analysis, particularly for working with structured data in tabular form.
2. **import matplotlib.pyplot as plt**: This line imports the Pyplot module from the Matplotlib library and aliases it as "plt." Matplotlib is a widely used library for creating static, animated, and interactive visualizations in Python.
3. **import seaborn as sns**: This line imports the Seaborn library, which is built on top of Matplotlib and provides a higher-level interface for creating attractive and informative statistical graphics. Seaborn is often used to enhance the aesthetics of Matplotlib plots.
4. **from collections import Counter**: This line imports the Counter class from the collections module. The Counter class is used for counting the occurrences of elements in an iterable, such as a list.

**Step -4**

```
df = pd.read_csv('/content/complaints.csv')
```

**Explanation-**

pd.read_csv('/content/complaints.csv'): This line of code uses Pandas to read a CSV file named "complaints.csv" located at the specified file path "/content/complaints.csv." The pd.read_csv() function is a Pandas function designed to read data from CSV files into a DataFrame, which is a two-dimensional tabular data structure.

**Step -5**

```
df.head()
```

**Explanation-**

- **df**: This is the name of the DataFrame that was previously created by reading data from a CSV file using Pandas, as seen in the previous code snippet. The DataFrame is typically a tabular data structure where rows represent observations or records, and columns represent attributes or variables.
- **.head()**: This is a method in Pandas that is applied to a DataFrame. When you call **.head()**, it returns the first few rows of the DataFrame. By default, it returns the first 5 rows, but you can specify a different number of rows to display by passing an integer argument inside the parentheses.

## Step -6

```
# Step 2: Text Pre-Processing
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import LabelEncoder
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import PorterStemmer
```

**Explanation-**

1. **from sklearn.model_selection import train_test_split**: This line imports the **train_test_split** function from the Scikit-learn library (**sklearn**). **train_test_split** is commonly used to split a dataset into training and testing subsets, which is essential for model training and evaluation.
2. **from sklearn.feature_extraction.text import TfidfVectorizer**: This line imports the **TfidfVectorizer** class from Scikit-learn. **TfidfVectorizer** is used to convert a collection of raw text documents into a numerical feature matrix, where each feature represents the importance of a word in the document using TF-IDF (Term Frequency-Inverse Document Frequency) weighting.
3. **from sklearn.preprocessing import LabelEncoder**: This line imports the **LabelEncoder** class from Scikit-learn. **LabelEncoder** is used for encoding categorical target labels (class labels) into numerical values, which is necessary for machine learning algorithms that require numerical inputs.

4. **import nltk**: This line imports the Natural Language Toolkit (NLTK), which is a popular Python library for NLP tasks. NLTK provides various tools and resources for working with human language data.
5. **from nltk.corpus import stopwords**: This line imports the NLTK corpus of stopwords. Stopwords are common words (e.g., "the," "and," "in") that are often removed from text data because they typically don't carry meaningful information for NLP tasks.
6. **from nltk.tokenize import word_tokenize**: This line imports the **word_tokenize** function from NLTK. Tokenization is the process of splitting a text into individual words or tokens, and **word_tokenize** is used for this purpose.
7. **from nltk.stem import PorterStemmer**: This line imports the **PorterStemmer** class from NLTK. Stemming is a text normalization technique that reduces words to their base or root form (e.g., "jumping" to "jump") to improve text analysis.

**Step -7**

```
# Download NLTK stopwords (if not already downloaded)
nltk.download('stopwords')
nltk.download('punkt')
```

**Explanation-**

1. **nltk.download('stopwords')**: This line uses the **nltk.download()** function to download the NLTK corpus of stopwords. Stopwords are common words in a language (e.g., "the," "and," "in") that are often removed from text data during preprocessing because they typically don't carry meaningful information for natural language processing (NLP) tasks. Downloading the stopwords corpus is necessary if you plan to use NLTK's predefined stopwords in your text preprocessing.
2. **nltk.download('punkt')**: This line uses the **nltk.download()** function to download the NLTK Punkt tokenizer models. The Punkt tokenizer is a pre-trained unsupervised machine learning model for tokenizing text into sentences and words. Downloading the Punkt tokenizer is important if you intend to use NLTK's tokenization functionality, specifically the **word_tokenize** function.

## Step -8

```
# Preprocess the text data
X = df['Consumer complaint narrative'].fillna('')  # Assuming text data is in 'Consumer complaint narrative'
y = df['Product']
```

**Explanation-**

1. X = df['Consumer complaint narrative'].fillna(''): This line extracts the column named "Consumer complaint narrative" from your DataFrame df and assigns it to the variable X. This column presumably contains the text data you want to preprocess. The fillna('') part is used to fill any missing (NaN) values in the selected column with empty strings (''). This step ensures that all entries in the "Consumer complaint narrative" column are strings, which is necessary for text processing.
2. y = df['Product']: This line extracts the column named "Product" from your DataFrame df and assigns it to the variable y. This column typically contains the target labels or categories associated with the text data in the "Consumer complaint narrative" column. These labels are used for supervised machine learning, where you aim to predict the product category based on the consumer complaints.

## Step -9

```
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

**Explanation-**

1. X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42): This line uses the train_test_split function from Scikit-learn (sklearn) to split your data into training and testing sets. Let's break down the parameters and what this line does:
    - X: This is the feature data, typically containing the independent variables or input data (in your case, the text data).
    - y: This is the target data, containing the labels or categories you want to predict (in your case, the product categories associated with the text data).
    - test_size=0.2: This parameter specifies the proportion of the data that should be allocated for the testing set. In this case, it's set to 0.2, which means 20% of the data will be used for testing, and the remaining 80% will be used for training.
    - random_state=42: This parameter is used to set a random seed for the data splitting process. Setting a seed ensures that the data split is reproducible, meaning if you run the code multiple times with the same seed (42 in this case), you'll get the same split every time. This is important for consistent results during model evaluation and testing.

After running this line of code, the data will be split into the following variables:

- X_train: This will contain the training feature data (typically the majority of your data).
- X_test: This will contain the testing feature data (used to evaluate the model's performance).
- y_train: This will contain the corresponding training target labels.
- y_test: This will contain the corresponding testing target labels.

**Step -10**

```
# Text Preprocessing
stop_words = set(stopwords.words('english'))
stemmer = PorterStemmer()
```

**Explanation-**

1. stop_words = set(stopwords.words('english')): This line of code sets up a set of stopwords for the English language. Let's break down the components:
    - nltk.corpus.stopwords.words('english'): This part of the code uses NLTK (Natural Language Toolkit) to access a predefined list of stopwords in the English language. Stopwords are common words (e.g., "the," "and," "in") that are often removed from text data because they typically don't carry meaningful information for NLP tasks.

- set(...): The result of stopwords.words('english') is a list of English stopwords. By wrapping it in set(...), you convert this list into a set. Using a set data structure provides faster membership checking, which is beneficial when you later check if words are stopwords.
- stop_words: The resulting set of English stopwords is stored in the variable stop_words, making it available for use in further text preprocessing.

2. stemmer = PorterStemmer(): This line of code initializes an instance of the Porter Stemmer, which is a stemming algorithm provided by NLTK.
    - Stemming is a text normalization technique that reduces words to their base or root form. For example, it might convert "running," "ran," and "runner" into the common root "run." This helps reduce the dimensionality of the data and can improve text analysis by treating similar words as the same.
    - PorterStemmer() creates an instance of the Porter Stemmer, which you can later use to apply stemming to individual words in your text data.

## Step -11

```
def preprocess(text):
  text="".join(word.lower() for word in text)
  tokens=word_tokenize(text)
  text= [stemmer.stem(word) for word in tokens if word not in stop_words]
  return text
```

**Explanation-**

1. **text = "".join(word.lower() for word in text)**: This line of code converts all the characters in the **text** to lowercase using **.lower()**. This step ensures that the text is consistently in lowercase, which is a common practice in text preprocessing to treat words in a case-insensitive manner.
2. **tokens = word_tokenize(text)**: This line tokenizes the text into individual words or tokens using the **word_tokenize** function from NLTK. Tokenization is the process of splitting a sentence or a piece of text into its constituent words or tokens.
3. **text = [stemmer.stem(word) for word in tokens if word not in stop_words]**: This line performs two important operations:
    - For each word in the **tokens** list, it applies stemming using the Porter Stemmer (**stemmer.stem(word)**). Stemming reduces words to their base or root form. For example, it might convert "running" to "run."
    - It checks if the word is not in the **stop_words** set (previously defined). If a word is in the **stop_words** set, it is removed

from the list. This step helps remove common and uninformative words from the text.

4. **return text**: Finally, the preprocessed list of words (tokens) is returned as the output of the **preprocess** function.

**Step -12**

```
# Encode target labels using the LabelEncoder
encoder = LabelEncoder()
y_train_encoded = encoder.fit_transform(y_train)

# Visualize the distribution of target classes (y_train_encoded)
plt.figure(figsize=(10, 6))
sns.countplot(y_train_encoded, palette='Set3')
plt.title('Distribution of Target Classes (y_train_encoded)')
plt.xlabel('Encoded Product Labels')
plt.ylabel('Count')
plt.show()
```

**Explanation-**

1. encoder = LabelEncoder(): This line initializes an instance of the LabelEncoder class from Scikit-learn (sklearn). The LabelEncoder is used for encoding categorical labels into numerical values. In your case, it's used to encode the "Product" labels into numerical form.
2. y_train_encoded = encoder.fit_transform(y_train): This line uses the fit_transform method of the LabelEncoder object (encoder) to transform the categorical target labels y_train into numerical values. The transformed values are stored in the variable y_train_encoded.

- fit_transform both fits the encoder to the unique labels in y_train and transforms the labels into integers. Each unique label is assigned a unique integer value.

3. The following code is used to visualize the distribution of the encoded target classes:
    - plt.figure(figsize=(10, 6)): This line creates a Matplotlib figure with a specific size to control the dimensions of the plot.
    - sns.countplot(y_train_encoded, palette='Set3'): This line uses Seaborn's countplot function to create a bar plot that displays the counts of each unique encoded label in y_train_encoded. The palette parameter specifies the color palette to use for the bars.
    - plt.title('Distribution of Target Classes (y_train_encoded)'): This line sets the title of the plot to describe what it displays.
    - plt.xlabel('Encoded Product Labels'): This line sets the label for the x-axis, indicating that the x-axis represents the encoded product labels.
    - plt.ylabel('Count'): This line sets the label for the y-axis, indicating that the y-axis represents the count of each encoded label.
    - plt.show(): This line displays the plot.

## Output-


Distribution of Target Classes (y_train_encoded)
1e6
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Count
0
Encoded Product Labels

## Step -13

```
 #Visualize the distribution of text lengths in X_train
text_lengths = [len(text.split()) for text in X_train]
plt.figure(figsize=(10, 6))
sns.histplot(text_lengths, bins=20, kde=True, color='skyblue')
plt.title('Distribution of Text Lengths in X_train')
plt.xlabel('Text Length (Number of Words)')
plt.ylabel('Count')
plt.show()
```

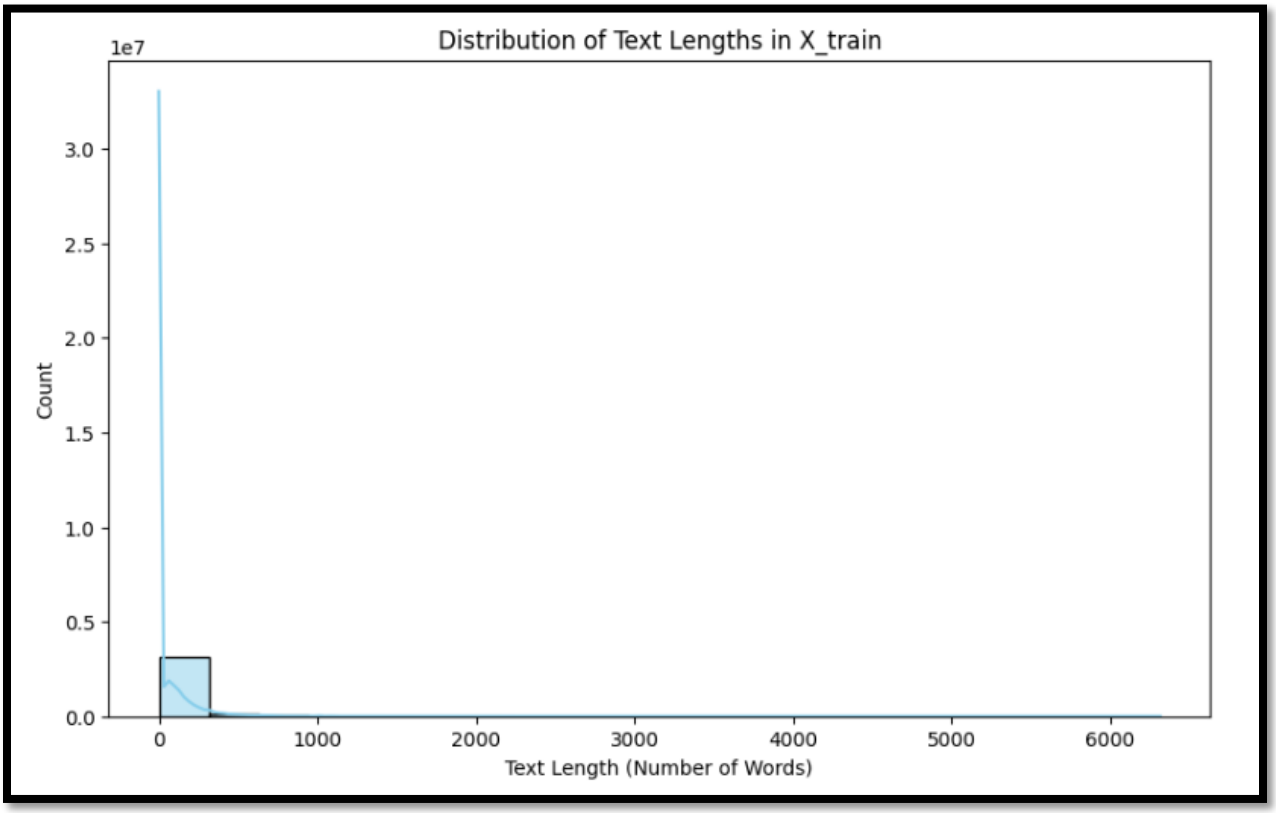
**Explanation-**

1. **text_lengths = [len(text.split()) for text in X_train]**: This line calculates the length of each text entry in the **X_train** dataset. Specifically, it does the following:
   - **for text in X_train**: It iterates through each text entry in the **X_train** dataset.
   - **len(text.split())**: For each text entry, it splits the text into words using **.split()**, and then it calculates the length (i.e., the number of words) in that text entry using **len()**. The result is a list (**text_lengths**) containing the number of words in each text entry in **X_train**.
2. **plt.figure(figsize=(10, 6))**: This line creates a Matplotlib figure with a specific size to control the dimensions of the plot. The **figsize** parameter sets the width and height of the figure in inches.
3. **sns.histplot(text_lengths, bins=20, kde=True, color='skyblue')**: This line uses Seaborn's **histplot** function to create a histogram of the distribution of text lengths in the **text_lengths** list. Here's what each parameter does:

- **text_lengths**: The data to be plotted, which is the list of text lengths.
- **bins=20**: This parameter specifies the number of bins or intervals in the histogram. In this case, it's set to 20, meaning the histogram will have 20 bars.
- **kde=True**: It overlays a kernel density estimate (KDE) plot on top of the histogram, providing a smoothed representation of the distribution.
- **color='skyblue'**: This sets the color of the bars and KDE plot in the histogram.

4. **plt.title('Distribution of Text Lengths in X_train')**: This line sets the title of the plot, describing what it displays.
5. **plt.xlabel('Text Length (Number of Words)')**: This line sets the label for the x-axis, indicating that the x-axis represents the number of words in the text.
6. **plt.ylabel('Count')**: This line sets the label for the y-axis, indicating that the y-axis represents the count of text entries.
7. **plt.show()**: This line displays the plot.

Output-


Distribution of Text Lengths in X_train
1e7
3.0
2.5
2.0
1.5
1.0
0.5
0.0
Count
0
1000
2000
3000
4000
5000
6000
Text Length (Number of Words)

## Step -14

```
# Visualize the most common words after preprocessing
def plot_most_common_words(corpus, num_words=10):
    flat_list = [word for sublist in corpus for word in sublist]
    word_counts = Counter(flat_list)
    most_common_words = word_counts.most_common(num_words)

    plt.figure(figsize=(10, 6))
    sns.barplot(x=[word[0] for word in most_common_words], y=[word[1] for word in most_common_words], palette='viridis')
    plt.title(f'Most Common {num_words} Words After Preprocessing')
    plt.xlabel('Words')
    plt.ylabel('Frequency')
    plt.xticks(rotation=45)
    plt.show()
```

## Explanation-

1. **def plot_most_common_words(corpus, num_words=10):**: This line defines a function called **plot_most_common_words** that takes two arguments:
    - **corpus**: This argument is expected to be a list of lists, where each inner list represents a document or a piece of text that has been preprocessed and tokenized.
    - **num_words=10**: This argument is optional and has a default value of 10. It specifies the number of most common words to display in the visualization.
2. **flat_list = [word for sublist in corpus for word in sublist]**: This line flattens the list of lists (**corpus**) into a single flat list called **flat_list**. It concatenates all the words from the preprocessed documents into a single list.
3. **word_counts = Counter(flat_list)**: This line uses the **Counter** class from the **collections** module to count the frequency of each word in the **flat_list**. The result is stored in the **word_counts** dictionary, where keys are unique words, and values are their frequencies.

4. **most_common_words = word_counts.most_common(num_words)**: This line retrieves the **num_words** most common words and their frequencies from the **word_counts** dictionary. The result is a list of tuples, where each tuple contains a word and its frequency.
5. The following code is used to create a bar plot to visualize the most common words:
    - **plt.figure(figsize=(10, 6))**: This line creates a Matplotlib figure with a specific size to control the dimensions of the plot.
    - **sns.barplot(x=[word[0] for word in most_common_words], y=[word[1] for word in most_common_words], palette='viridis')**: This line uses Seaborn's **barplot** function to create a bar plot. It specifies the x-axis values as the words (**[word[0] for word in most_common_words]**) and the y-axis values as their frequencies (**[word[1] for word in most_common_words]**). The **palette** parameter sets the color palette for the bars.
    - **plt.title(f'Most Common {num_words} Words After Preprocessing')**: This line sets the title of the plot to describe what it displays, including the number of most common words.
    - **plt.xlabel('Words')**: This line sets the label for the x-axis, indicating that the x-axis represents words.
    - **plt.ylabel('Frequency')**: This line sets the label for the y-axis, indicating that the y-axis represents word frequencies.
    - **plt.xticks(rotation=45)**: This line rotates the x-axis labels by 45 degrees for better readability if the words are long.

- **plt.show()**: This line displays the plot.

**Step -15**

```
vectorizer=TfidfVectorizer(analyzer=preprocess)
```

**Explanation-**

- **vectorizer**: This is a variable that stores an instance of the **TfidfVectorizer** class. In machine learning and natural language processing (NLP), TF-IDF (Term Frequency-Inverse Document Frequency) vectorization is a technique used to convert a collection of text documents into numerical feature vectors, where each feature represents the importance of a word in a document relative to a collection of documents.
- **TfidfVectorizer**: This is a class provided by Scikit-learn (**sklearn**) that implements the TF-IDF vectorization process. It takes various parameters to control how the vectorization is performed.
- **analyzer=preprocess**: This parameter specifies the analyzer function to be used during vectorization. In this case, you've provided the custom function **preprocess** as the analyzer. The **analyzer** parameter allows you to define a custom preprocessing function that will be applied to each document before vectorization.

## Step -16

```
vectorizer.fit(X_train)
```

```
TfidfVectorizer
TfidfVectorizer(analyzer=<function preprocess at 0x7e9d14afbf40>)
```

**Explanation-**

- **vectorizer**: This is the TF-IDF vectorizer object that you previously initialized with custom text preprocessing settings using the **TfidfVectorizer** class. It's responsible for converting text data into numerical feature vectors using the TF-IDF transformation.
- **.fit(X_train)**: This is a method call on the **vectorizer** object. When you call **.fit()** with the training data **X_train**, the vectorizer learns and computes the necessary statistics and parameters from the training data. Specifically:
    - It builds a vocabulary: The vectorizer identifies all unique words (or tokens) in the training data and creates a vocabulary. Each word in the vocabulary corresponds to a feature in the vectorized representation.
    - It computes the document frequency (DF) for each word: The vectorizer counts how many documents in the training data contain each word. This information is used to compute the IDF (Inverse Document Frequency) component of the TF-IDF score.
    - It computes the term frequency-inverse document frequency (TF-IDF) values for each word in each document: The TF-IDF values represent the importance of each word in

each document relative to the entire training corpus. It takes into account both the frequency of the word within a document (TF) and how unique the word is across all documents (IDF).

**Step -17**

```
X_train=vectorizer.transform(X_train)
X_test=vectorizer.transform(X_test)
```

**Explanation-**

is used to transform your text data from the training and testing sets (X_train and X_test) into numerical feature vectors using the TF-IDF vectorizer (vectorizer) that you previously fitted to the training data. Let's break down what this code does:

- X_train: This line transforms the training text data (X_train) into numerical feature vectors using the trained TF-IDF vectorizer (vectorizer). After this line of code, X_train will contain the TF-IDF representations of the text data from the training set.
- X_test: Similarly, this line transforms the testing text data (X_test) into numerical feature vectors using the same trained TF-IDF vectorizer (vectorizer). After this line of code, X_test will contain the TF-IDF representations of the text data from the testing set.

TF-IDF vectorization converts text data into a numerical format that machine learning models can work with. Each document in the dataset is represented as a numerical vector where each element corresponds to a word or term in the vocabulary, and the value in each element is

the TF-IDF score of that word in the document. These TF-IDF vectors can then be used as input features for machine learning models.

**Step -18**

```
encoder=LabelEncoder()
y_train=encoder.fit_transform(y_train)
y_test=encoder.fit_transform(y_test)
```

**Explanation-**

1. **encoder = LabelEncoder()**: This line initializes an instance of the **LabelEncoder** class from Scikit-learn (**sklearn**). The **LabelEncoder** is commonly used for encoding categorical labels into numerical values. In your case, it's used to encode the "Product" labels into numerical form.
2. **y_train = encoder.fit_transform(y_train)**: This line uses the **fit_transform** method of the **LabelEncoder** object (**encoder**) to transform the categorical target labels in the training dataset (**y_train**) into numerical values. The transformed values are stored in the variable **y_train**.
    - **fit_transform** both fits the encoder to the unique labels in **y_train** (i.e., it learns the mapping between unique labels and numerical values) and transforms the labels into integers. Each unique label is assigned a unique integer value.
3. **y_test = encoder.fit_transform(y_test)**: Similarly, this line uses the **fit_transform** method to transform the categorical target

labels in the testing dataset (**y_test**) into numerical values. The transformed values are stored in the variable **y_test**.

**Step -19**

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

**Explanation-**

1. from sklearn.linear_model import LogisticRegression: This line imports the LogisticRegression class from Scikit-learn (sklearn). Logistic regression is a common classification algorithm used for binary and multiclass classification tasks.
2. clf = LogisticRegression(): This line initializes an instance of the LogisticRegression classifier. The variable clf is now an instance of the logistic regression model that you can use for training and prediction.
3. clf.fit(X_train, y_train): This line trains the logistic regression model (clf) using the training data:
    - X_train: This is the feature matrix containing the TF-IDF representations of the training text data. It is used as the input features for training.
    - y_train: This is the target variable containing the encoded labels (numerical values) corresponding to the product categories. It is used as the target variable for training.

When you call .fit() with these arguments, the logistic regression model learns to fit a decision boundary that separates the input features (X_train) into different classes represented by the target labels (y_train). This is the training phase of the model.

4. y_pred = clf.predict(X_test): After training the logistic regression model, this line uses the trained model to make predictions on the testing data:
    - X_test: This is the feature matrix containing the TF-IDF representations of the testing text data. It is used as the input features for prediction.

The .predict() method takes the feature matrix as input and returns predicted class labels for each example in the testing data. These predictions are stored in the variable y_pred.

**Step -20**

```
from sklearn.metrics import classification_report
print(classification_report(y_test,y_pred))
```

**Explanation-**

1. from sklearn.metrics import classification_report: This line imports the classification_report function from Scikit-learn's sklearn.metrics module. The classification_report function is used to compute and print various classification metrics for a classification problem.
2. print(classification_report(y_test, y_pred)): This line calls the classification_report function to generate a classification report

based on the actual target labels (y_test) and the predicted labels (y_pred) from your classifier. Here's what the function does:

- y_test: This argument is the true target labels for the testing data. It represents the ground truth labels that you want to compare the model's predictions against.
- y_pred: This argument is the predicted target labels for the testing data, which were generated by your logistic regression model. These are the labels that you want to evaluate.

When you call classification_report with these arguments, it calculates and prints several important classification metrics, including but not limited to:

- Precision: The proportion of true positive predictions among all positive predictions. It measures the model's accuracy when predicting positive cases.
- Recall (Sensitivity): The proportion of true positive predictions among all actual positive cases. It measures the model's ability to identify all positive cases.
- F1-score: The harmonic mean of precision and recall, providing a balance between the two metrics. It's a useful metric when dealing with imbalanced datasets.
- Support: The number of samples in each class (product category).
- Accuracy: The overall accuracy of the model's predictions.

Output-

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.39 | 0.02 | 0.04 | 17336 |
| 1 | 0.68 | 0.33 | 0.44 | 36623 |
| 2 | 0.29 | 0.01 | 0.01 | 6363 |
| 3 | 0.46 | 0.02 | 0.03 | 18373 |
| 4 | 0.67 | 0.37 | 0.48 | 41473 |
| 5 | 0.52 | 0.02 | 0.03 | 28017 |
| 6 | 0.00 | 0.00 | 0.00 | 14473 |
| 7 | 0.61 | 0.98 | 0.75 | 433405 |
| 8 | 0.77 | 0.32 | 0.45 | 101727 |
| 9 | 0.00 | 0.00 | 0.00 | 7 |
| 10 | 0.72 | 0.37 | 0.49 | 11886 |
| 11 | 0.00 | 0.00 | 0.00 | 1072 |
| 12 | 0.86 | 0.25 | 0.39 | 76533 |
| 13 | 0.00 | 0.00 | 0.00 | 197 |
| 14 | 0.00 | 0.00 | 0.00 | 1138 |
| 15 | 0.46 | 0.15 | 0.23 | 6085 |
| 16 | 0.00 | 0.00 | 0.00 | 64 |
| 17 | 0.00 | 0.00 | 0.00 | 800 |
| 18 | 0.86 | 0.35 | 0.50 | 15159 |
| 19 | 0.59 | 0.26 | 0.36 | 9539 |
| 20 | 0.00 | 0.00 | 0.00 | 7 |
| accuracy | | | 0.63 | 820277 |
| macro avg | 0.38 | 0.16 | 0.20 | 820277 |
| weighted avg | 0.64 | 0.63 | 0.56 | 820277 |

## Step -21

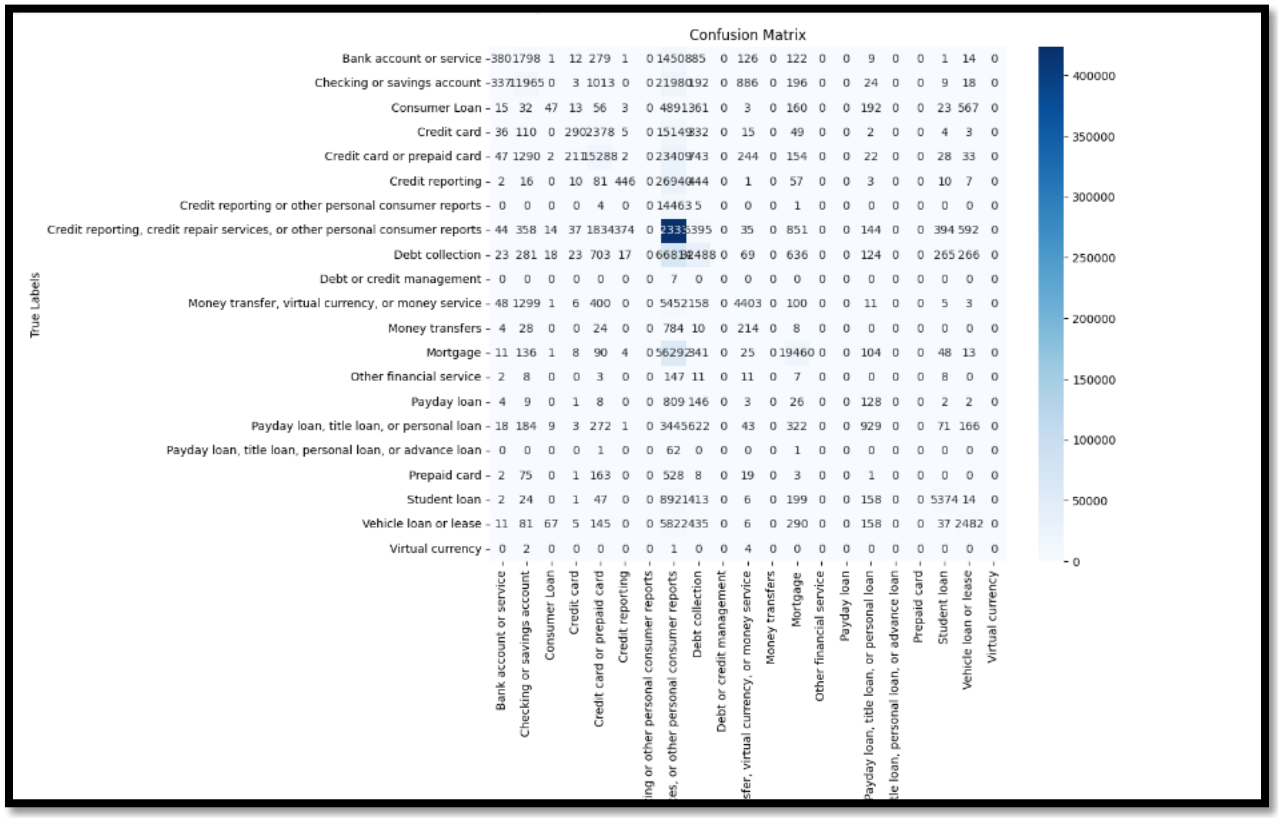
```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix

# Calculate the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Create a heatmap for the confusion matrix
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=encoder.classes_, yticklabels=encoder.classes_)
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.show()
```

**Explanation-**

1. import matplotlib.pyplot as plt: This line imports the Matplotlib library, which is commonly used for creating various types of plots and visualizations.
2. from sklearn.metrics import confusion_matrix: This line imports the confusion_matrix function from Scikit-learn's sklearn.metrics module. The confusion_matrix function computes a confusion matrix for a classification problem, which is a table that summarizes the classification results and helps evaluate the performance of a model.
3. cm = confusion_matrix(y_test, y_pred): This line calculates the confusion matrix by comparing the true target labels (y_test) with the predicted labels (y_pred) from your classification model. The result is stored in the variable cm, which is a 2D array.
4. plt.figure(figsize=(10, 8)): This line creates a Matplotlib figure with specific dimensions (10 inches in width and 8 inches in height) to control the size of the heatmap plot.

5. sns.heatmap(cm, annot=True, fmt="d", cmap="Blues", xticklabels=encoder.classes_, yticklabels=encoder.classes_):
    - sns.heatmap: This line uses Seaborn's heatmap function to create a heatmap visualization of the confusion matrix.
    - cm: It specifies the confusion matrix that you want to visualize.
    - annot=True: This parameter indicates that you want to annotate the cells of the heatmap with the actual numerical values from the confusion matrix.
    - fmt="d": This parameter specifies the format for displaying the numerical values as integers (e.g., 0, 1, 2).
    - cmap="Blues": This parameter sets the color map for the heatmap. In this case, it uses shades of blue.
    - xticklabels=encoder.classes_, yticklabels=encoder.classes_): These parameters specify the labels for the x-axis and y-axis of the heatmap. encoder.classes_ contains the class labels for your classification problem, and it sets the tick labels for both axes.
6. plt.xlabel('Predicted Labels'): This line sets the label for the x-axis, indicating that it represents the predicted labels.
7. plt.ylabel('True Labels'): This line sets the label for the y-axis, indicating that it represents the true (actual) labels.
8. plt.title('Confusion Matrix'): This line sets the title of the heatmap to "Confusion Matrix."
9. plt.show():

# Output-


Confusion Matrix
True Labels
Bank account or service
Checking or savings account
Consumer Loan
Credit card
Credit card or prepaid card
Credit reporting
Credit reporting or other personal consumer reports
Credit reporting, credit repair services, or other personal consumer reports
Debt collection
Debt or credit management
Money transfer, virtual currency, or money service
Money transfers
Mortgage
Other financial service
Payday loan
Payday loan, title loan, or personal loan
Payday loan, title loan, personal loan, or advance loan
Prepaid card
Student loan
Vehicle loan or lease
Virtual currency
400000
350000
300000
250000
200000
150000
100000
50000
0