Dataset Description

The **Sentiment140** dataset is a collection of 1.6 million tweets, which are labeled with sentiment labels. It was originally created to perform sentiment analysis on Twitter data. The dataset includes two primary sentiment categories: **positive** and **negative**. The **neutral** sentiment label is excluded for this task to focus on binary classification. Each tweet is labeled as either positive (4) or negative (0), based on its content, making it suitable for training machine learning models to predict sentiment based on text data.

The dataset includes 6 columns, including:

1. **target**: Sentiment label (0 for negative, 4 for positive).
2. **id**: A unique identifier for each tweet.
3. **date**: The timestamp when the tweet was posted.
4. **flag**: A column indicating if the tweet was flagged.
5. **user**: The user who posted the tweet.
6. **text**: The actual tweet content.

This dataset is valuable for training sentiment analysis models and evaluating their performance. In this project, the dataset is further preprocessed to ensure balanced sentiment classes for accurate model training.

Dataset link : https://www.kaggle.com/datasets/kazanova/sentiment140

Project Implementation Detail

Title of Task 1: Project Setup and Library Imports

In this task, we set up the necessary libraries and dependencies required for the sentiment analysis project. This includes:

1. **Data Handling**: Importing libraries like numpy and pandas for efficient data manipulation.
2. **Text Preprocessing**: Using **NLTK** (Natural Language Toolkit) for tasks such as tokenization, stopword removal, stemming, and lemmatization.
3. **Machine Learning**: Importing modules from **Scikit-learn** for vectorization, splitting the dataset, and applying the **Naïve Bayes** classification algorithm.
4. **Visualization**: Importing matplotlib for visualizing data and model performance.

Additionally, we download the necessary datasets from **NLTK** to ensure the tools for text processing are available.

Code for Task 1:

```
[321...
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay, accuracy_score
import seaborn as sns
import nltk
from nltk.corpus import stopwords, wordnet
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.tokenize import word_tokenize
nltk.download('stopwords'); nltk.download('punkt'); nltk.download('wordnet'); nltk.download('omw-1.4')
import re
import string
```

Sample output for task 1 :

```
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\ASUS\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\ASUS\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to
[nltk_data]     C:\Users\ASUS\AppData\Roaming\nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package omw-1.4 to
[nltk_data]     C:\Users\ASUS\AppData\Roaming\nltk_data...
```

Code Description of Task 1:

1. **Library Imports**:
    - **Data Handling**: We import numpy and pandas to load, manipulate, and analyze the dataset.
    - **Visualization**: matplotlib.pyplot and seaborn are used for plotting graphs, including confusion matrices and accuracy metrics.
    - **Text Preprocessing**: The nltk library is imported for handling text processing tasks. We use **s**topwords for filtering out common, non-informative words, **word_tokenize** for splitting text into individual words, **PorterStemmer** and **WordNetLemmatizer** for reducing words to their base forms.
    - **Machine Learning**: **train_test_split** is used to split the dataset into training and test sets, **TfidfVectorizer** converts the text into numerical vectors, and **MultinomialNB** applies the Naïve Bayes classification algorithm.
2. **Downloading Resources**:
    - We use nltk.download() to download the required datasets:
        - **stopwords** for removing common words like "the", "is", "in".
        - **punkt** for tokenization.
        - **wordnet** for lemmatization (reducing words like "better" to "good").
        - **omw-1.4** for multilingual support.

Title of Task 2: Dataset Loading and Preprocessing

In this task, we are preparing the dataset for sentiment classification by addressing dataset imbalance and cleaning the data. First, we load the **Sentiment140** dataset from a CSV file and filter it to include only the relevant sentiment labels: 0 (negative) and 4 (positive), while excluding the neutral sentiment (2). This ensures that we focus on the primary sentiments for our binary classification task.

Next, we balance the dataset by randomly sampling an equal number of positive and negative tweets, specifically **100,000 tweets from each class**. This step is important to avoid any bias towards one sentiment and ensures the model is trained on an equal distribution of positive and negative examples. After sampling, we reset the DataFrame's index to maintain a clean structure. Finally, the processed dataset is saved to a new CSV file for future use, and the numeric sentiment labels are mapped to more readable string labels: 0 becomes 'negative', and 4 becomes 'positive'.

Code for Task 2:

```
[449]: columns = ["target", "id", "date", "flag", "user", "text"]
df = pd.read_csv("training.1600000.processed.noemoticon.csv", encoding='latin-1', names=columns)


df = df[df['target'].isin([0, 4])]


sample_per_class = 100000

df = pd.concat([
    df[df['target']==0].sample(sample_per_class, random_state=42),
    df[df['target']==4].sample(sample_per_class, random_state=42)
])


df = df.reset_index(drop=True)


df.to_csv("balanced_sentiment140_no_neutral.csv", index=False)

print("new data set for  my project ")

label_map = {0:'negative',2:'neutral',4:'positive'}
df['label'] = df['target'].map(label_map).fillna(df['target'])
print(df[['text','label']].head())
print(df[['text','label']].tail())
```

Sample output for task 2 :

```
new data set for  my project
                                                text     label
0  @xnausikaax oh no! where did u order from? tha...  negative
1  A great hard training weekend is over.  a coup...  negative
2  Right, off to work  Only 5 hours to go until I...  negative
3                  I am craving for japanese food   negative
4  Jean Michel Jarre concert tomorrow  gotta work...  negative
                                                text     label
199995  @kimberlykeith Awww thanks  That made me feel ...  positive
199996  @callkathy I just started using this on my iPh...  positive
199997  @tferriss Hi Tim, what are your thoughts of Bu...  positive
199998  @pastorjpruitt We have authority over devils (...  positive
199999  @greaggedeanman, the one with ''saving the wor...  positive
```

Code Description of Task 2:

1. **Loading the Dataset**:
   - We use pd.read_csv() to load the dataset from the CSV file and define custom column names for easier manipulation.
2. **Filtering Data**:
   - We use df[df['target'].isin([0, 4])] to filter the DataFrame and keep only rows with sentiment labels 0 (negative) and 4 (positive), excluding neutral sentiment (2).
3. **Balancing the Dataset**:
   - To balance the dataset, we use the sample() function to randomly select 100,000 positive and 100,000 negative tweets. These sampled tweets are combined using pd.concat().

4. **Resetting the Index**:
    - After sampling, we use df.reset_index(drop=True) to reset the index and remove any gaps in the DataFrame index.
5. **Saving the Processed Data**:
    - We save the balanced dataset into a new CSV file using df.to_csv(). This ensures that we have a clean and balanced dataset ready for further analysis or model training.
6. **Mapping Sentiment Labels**:
    - The label_map dictionary is used to convert numeric sentiment labels (0, 4) to more meaningful string labels ('negative', 'positive').
7. **Displaying the Data**:
    - The head() and tail() functions are used to display the first and last few rows of the dataset to confirm that the sentiment labels are correctly mapped.

Title of Task 3: Sentiment Distribution Visualization

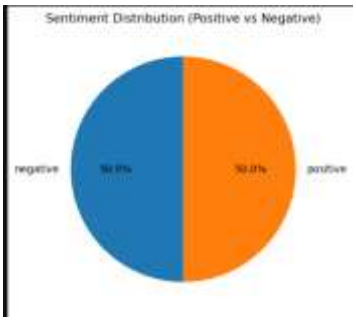
In this task, we aim to visualize the distribution of sentiments (positive vs. negative) in the dataset. By using a pie chart, we can easily observe the proportion of positive and negative tweets, which helps in understanding the dataset's balance. This task is important to ensure that the sentiment labels are equally distributed and that no class is overrepresented, which could impact model performance.

Code for Task 3:

```
counts = df['label'].value_counts()

plt.figure(figsize=(5,5))
plt.pie(
    counts,
    labels=counts.index,
    autopct='%1.1f%%',
    startangle=90
)
plt.title('Sentiment Distribution (Positive vs Negative)')
plt.show()
```

Sample Output of Task 3:



Code Description of Task 3:

1. **Counting the Sentiment Labels**:
    - We use df['label'].value_counts() to count the number of positive and negative sentiments in the dataset.
2. **Plotting the Pie Chart**:
    - The plt.pie() function is used to create a pie chart. The counts are passed to represent the proportions, and counts.index provides the labels for the chart (i.e., positive and negative).
    - The autopct='%1.1f%%' argument displays the percentage of each category in the chart.
    - The startangle=90 ensures that the pie chart starts from a specific angle for better visibility.
3. **Displaying the Chart**:
    - Finally, plt.show() displays the pie chart, allowing us to visually confirm the sentiment distribution in the dataset.

Title of Task 4: Handling Missing Values

In this task, we address missing values in the dataset. It is important to ensure that there are no missing or null values in the text and target columns, as these could interfere with training a machine learning model. We check for any null values in the dataset and drop rows where either the text or target column has missing values. This step ensures that the dataset is clean and complete, and ready for model training.

Code for Task 4:

Sample Output of Task 4:

Code Description of Task 4:

1. **Checking for Null Values**:
    - The df.isnull().sum() function is used to count the number of missing (null) values in each column of the DataFrame. The result is displayed using print().

2. **Dropping Null Values**:
    - The dropna() function is used to remove rows where either the text or target column contains null values. This ensures that we only keep rows with valid text and sentiment labels.
3. **Final Row Count**:
    - After dropping the rows with missing values, the final number of rows in the dataset is printed using len(df), ensuring that the dataset is clean and ready for use.

Title of Task 5: Text Lowercasing

In this task, we focus on standardizing the text by converting all characters to lowercase. This step is crucial because, in text analysis, words like "Happy" and "happy" should be considered the same. By transforming all text to lowercase, we eliminate any case-based differences between words. This ensures uniformity in the dataset and helps in reducing the complexity for the model.

Code for Task 5:

```
df['text_lower'] = df['text'].str.lower()
print (df['text_lower'].head())
```

Sample Output of Task 5:

```
0    @xnausikaax oh no! where did u order from? tha...
1    a great hard training weekend is over.  a coup...
2    right, off to work  only 5 hours to go until i...
3                      i am craving for japanese food
4    jean michel jarre concert tomorrow  gotta work...
Name: text_lower, dtype: object
```

Code Description of Task 5:

1. **Converting Text to Lowercase**:
    - We use df['text'].str.lower() to convert all the text in the text column to lowercase. The str.lower() method ensures that all characters in the text are changed to lowercase, making the text uniform.
2. **Storing the Result**:
    - The transformed text is stored in a new column called text_lower in the DataFrame.
3. **Displaying the Output**:
    - The head() function is used to display the first few rows of the text_lower column to confirm that the transformation was successful.

Title of Task 6: Text Cleaning

In this task, we clean the text data by removing unwanted elements such as URLs, mentions, hashtags, and non-alphabetic characters. This step is important because such elements may not contribute to sentiment analysis and could add noise to the model. The cleaning process will also ensure that the text is in a uniform format and ready for further preprocessing.

Code for Task 6:

```
def clean_text(text):
    text = re.sub(r"http\S+", "", text)
    text = re.sub(r"@\w+", "", text)
    text = re.sub(r"#\w+", "", text)
    text = re.sub(r"[^a-zA-Z\s]", "", text)
    text = re.sub(r"\s+", " ", text).strip()
    return text

df["clean_text"] = df["text_lower"].apply(clean_text)
print(df[['clean_text']].head())
```

Sample Output of Task 6:

```
                                          clean_text
0        oh no where did u order from thats horrible
1  a great hard training weekend is over a couple...
2  right off to work only hours to go until im fr...
3                    i am craving for japanese food
4  jean michel jarre concert tomorrow gotta work ...
```

Code Description of Task 6:

1. **Cleaning the Text**:
    - We define the clean_text() function that takes a string as input and applies several regular expression (re.sub) operations to clean the text:
        - r"http\S+": Removes URLs starting with "http".
        - r"@\w+": Removes mentions (e.g., @username).
        - r"#\w+": Removes hashtags (e.g., #hashtag).
        - r"[^a-zA-Z\s]": Removes non-alphabetic characters, keeping only letters and spaces.
        - r"\s+": Replaces multiple spaces with a single space and removes extra leading or trailing spaces.
2. **Applying the Cleaning Function**:
    - The apply(clean_text) function is used to apply this cleaning function to every row in the text_lower column.
    - The cleaned text is stored in a new column called clean_text.
3. **Displaying the Cleaned Data**:

- The head() function is used to display the first few rows of the cleaned text to confirm that the cleaning process was successful.

Title of Task 7: Removing Punctuation

In this task, we remove punctuation from the text to ensure that only the meaningful content is retained for sentiment analysis. Punctuation marks like commas, periods, exclamation points, and others do not contribute to the sentiment of the text and can be safely removed. This cleaning step helps in normalizing the text and prepares it for the next stages of processing.

Code for Task 7:

```
df['text_nopunct'] = df['clean_text'].apply(lambda x: re.sub(f"[{re.escape(string.punctuation)}]", "", x))

print(df[['text_nopunct']].head())
```

Sample Output of Task 7:

```
                                        text_nopunct
0        oh no where did u order from thats horrible
1  a great hard training weekend is over a couple...
2  right off to work only hours to go until im fr...
3                    i am craving for japanese food
4  jean michel jarre concert tomorrow gotta work ...
```

Code Description of Task 7:

1. **Removing Punctuation**:
   - We use re.sub() with a regular expression to remove all punctuation from the text. The string.punctuation constant provides a list of punctuation characters (e.g., ., ,, !, ?, etc.).
   - The re.escape() function ensures that all punctuation characters are treated as literal characters in the regular expression, rather than special regex symbols.
2. **Applying the Removal**:
   - The apply() function is used to apply the punctuation removal process to each row in the clean_text column.
   - The result is stored in a new column, text_nopunct.
3. **Displaying the Output**:
   - The head() function is used to display the first few rows of the cleaned text without punctuation.

Title of Task 8: Tokenization

In this task, we perform **tokenization** on the cleaned text, which involves splitting the text into individual words (tokens). Tokenization is a critical step in text preprocessing, as it converts the raw text into smaller, more manageable pieces (words), which can then be used for further analysis or modeling. This step helps break down the text into the fundamental building blocks, making it easier for machine learning models to process.

Code for Task 8:

```
df['tokens'] = df['text_nopunct'].apply(word_tokenize)
print(df[['tokens']].head())
```

Sample Output of Task 8:

```
                                                tokens
0  [oh, no, where, did, u, order, from, thats, ho...
1  [a, great, hard, training, weekend, is, over, ...
2  [right, off, to, work, only, hours, to, go, un...
3                [i, am, craving, for, japanese, food]
4  [jean, michel, jarre, concert, tomorrow, got, ...
```

Code Description of Task 8:

1. **Tokenization**:
   - We use the word_tokenize() function from the nltk library to split the text in the text_nopunct column into individual tokens (words).
   - This process transforms the text into a list of words, where each word becomes a separate token, making the text ready for further processing.
2. **Applying Tokenization**:
   - The apply() function is used to apply the word_tokenize() function to each row in the text_nopunct column.
   - The tokenized words are stored in a new column called tokens.
3. **Displaying the Output**:
   - The head() function is used to display the first few rows of the tokens column, which now contains lists of words from the original text.

Title of Task 9: Stopword Removal

In this task, we remove **stopwords** from the tokenized text. Stopwords are common words like "the", "is", "in", etc., that do not carry significant meaning and are typically removed in text preprocessing. Removing these words helps reduce the dimensionality of the text and improves the model's performance by focusing on more meaningful words. We apply this step after tokenization to clean up the text further.

```
stop_words = set(stopwords.words('english'))
df['tokens_nostop'] = df['tokens'].apply(lambda tokens: [word for word in tokens if word not in stop_words])
print(df[['tokens_nostop']].head())
```

Sample Output of Task 9:

```
                                              tokens_nostop
0                        [oh, u, order, thats, horrible]
1  [great, hard, training, weekend, couple, days,...
2                  [right, work, hours, go, im, free, xd]
3                               [craving, japanese, food]
4  [jean, michel, jarre, concert, tomorrow, got, ...
```

Code

Description of Task 9:

1. **Stopword List**:
   - We use stopwords.words('english') from the nltk library to load the list of common English stopwords. These are words that do not carry significant meaning for sentiment analysis and can be ignored.
2. **Removing Stopwords**:
   - We use the apply() function along with a lambda function to iterate through the tokenized words in the tokens column. For each list of tokens, we filter out any word that is present in the stop_words set.
   - The result is stored in a new column called tokens_nostop, which contains the tokenized text without the stopwords.
3. **Displaying the Output**:
   - The head() function is used to display the first few rows of the tokens_nostop column, showing the cleaned tokens after the stopwords have been removed.

Title of Task 10: Synonym Substitution

In this task, we perform **synonym substitution** to enhance the text data by replacing words with their synonyms. This technique helps in diversifying the text, making the model more robust to different word forms. We use **WordNet**, a lexical database in the nltk library, to find synonyms for each word in the tokenized text. If a synonym is found, we replace the word with the first lemma (base form) of its synonym. This step is useful to reduce redundancy and improve the generalization of the model.

Code for Task 10:

```python
def synonym_sub(tokens):
    out = []
    for w in tokens:
        syns = wordnet.synsets(w)
        if syns:
            lemmas = syns[0].lemmas()
            if lemmas:
                out.append(lemmas[0].name().replace('_',' '))
            else:
                out.append(w)
        else:
            out.append(w)
    return out

df['tok_syn'] = df['tokens_nostop'].apply(synonym_sub)
print(df[['tok_syn']].head())
```

Sample Output of Task 10:

```
                                                 tok_syn
0              [Ohio, uracil, order, thats, atrocious]
1  [great, difficult, training, weekend, couple, ...
2              [right, work, hours, go, im, free, xd]
3                           [craving, Japanese, food]
4  [jean, michel, jarre, concert, tomorrow, get, ...
```

Code Description of Task 10:

1. **Synonym Substitution**:
    - The synonym_sub() function iterates over each word in the tokenized text (tokens_nostop column).
    - For each word, it uses **WordNet** to find possible synonyms (synsets) and retrieves the first lemma (base form) of the first synonym set.
    - If no synonym is found, the original word is kept in the output.
2. **Applying Synonym Substitution**:
    - The apply() function applies the synonym_sub() function to each row in the tokens_nostop column.
    - The result is stored in a new column, tok_syn, which contains the tokens with substituted synonyms.
3. **Displaying the Output**:

- The head() function is used to display the first few rows of the tok_syn column, showing the tokens after synonym substitution.

Title of Task 11: Stemming

In this task, we perform **stemming** on the tokenized text. Stemming is the process of reducing words to their root form (e.g., "running" becomes "run"). This step helps in normalizing the text and reducing redundancy by grouping different forms of a word under a single root. We use the **Porter Stemmer** from the nltk library, which is widely used for stemming English words. By performing stemming, we ensure that the text is simplified and more uniform for sentiment analysis.

Code for Task 11:

```
stemmer = PorterStemmer()
df['tok_stem'] = df['tok_syn'].apply(lambda toks: [stemmer.stem(w) for w in toks])
df[['tok_stem']].head()
```

Sample Output of Task 11:

	tok_stem
0	[ohio, uracil, order, that, atroci]
1	[great, difficult, train, weekend, coupl, day,...
2	[right, work, hour, go, im, free, xd]
3	[crave, japanes, food]
4	[jean, michel, jarr, concert, tomorrow, get, t...

Code Description of Task 11:

1. **Porter Stemmer Initialization**:
   - We initialize the **PorterStemmer** from the nltk library, which is designed to remove common suffixes from English words (e.g., "ing", "es", "ed") to reduce them to their root form.
2. **Applying Stemming**:
   - The apply() function is used to apply a lambda function to each row in the tok_syn column. The lambda function iterates over the tokens (words) in the list and applies the stem() method to each word, reducing it to its root form.
   - The result is stored in a new column called tok_stem, which contains the stemmed tokens.

3. **Displaying the Output**:
    - The head() function is used to display the first few rows of the tok_stem column, showing the tokens after stemming.

Title of Task 12: Lemmatization

In this task, we perform **lemmatization** on the tokenized and stemmed text. Lemmatization is a more sophisticated text normalization process compared to stemming. It reduces words to their base form (or lemma) based on their part of speech (POS), such as nouns, verbs, and adjectives. This allows the model to process words like "better" and "good" as the same word, "good," while retaining meaningful distinctions. By applying lemmatization, we ensure that the text is normalized in a more linguistically accurate manner.

Code for Task 12:

```
lemm = WordNetLemmatizer()
def lemmatize_tokens(toks):
    out = []
    for w in toks:
        l = lemm.lemmatize(w, pos='n')
        l = lemm.lemmatize(l, pos='v')
        l = lemm.lemmatize(l, pos='a')
        out.append(l)
    return out
df['tok_lem'] = df['tok_stem'].apply(lemmatize_tokens)
df[['tok_lem']].head()
```

Sample Output of Task 12:

	tok_lem
0	[ohio, uracil, order, that, atroci]
1	[great, difficult, train, weekend, coupl, day,...
2	[right, work, hour, go, im, free, xd]
3	[crave, japan, food]
4	[jean, michel, jarr, concert, tomorrow, get, t...

Code Description of Task 12:

1. **WordNet Lemmatizer Initialization**:
   - The WordNetLemmatizer() is initialized from the nltk library. It is used to reduce words to their base forms (lemmas) based on their part of speech (POS).
2. **Lemmatization Function**:
   - The lemmatize_tokens() function processes each word in the list of tokens. For each word, it is lemmatized three times: first as a noun (pos='n'), then as a verb (pos='v'), and finally as an adjective (pos='a'), in order to cover different possible word forms.
3. **Applying Lemmatization**:
   - The apply() function is used to apply the lemmatize_tokens() function to each row in the tok_stem column. The lemmatized tokens are stored in a new column called tok_lem.
4. **Displaying the Output**:
   - The head() function is used to display the first few rows of the tok_lem column, showing the tokens after lemmatization.

Title of Task 13: Final Text Preparation and Label Extraction

In this task, we prepare the final text and the corresponding labels for the machine learning model. First, we **join** the lemmatized tokens into a single string for each tweet. This step ensures that the text is in the required format for the model, with each tweet represented as a single string of words. Then, we extract the labels (positive/negative sentiment) and store them as a separate variable. Finally, we confirm the number of samples and check the distribution of sentiment labels in the dataset to ensure we have the correct structure for model training.

Code for Task 13:

```
df['final_text'] = df['tok_lem'].apply(lambda toks: ' '.join(toks))
X = df['final_text'].values
y = df['label'].values
len(X), len(y), pd.Series(y).value_counts()
```

Sample Output of Task 13:

```
(200000,
 200000,
 negative    100000
 positive    100000
 Name: count, dtype: int64)
```

This confirms that the dataset has **200,000 samples** with an equal distribution of positive and negative labels.

Code Description of Task 13:

1. **Final Text Preparation**:
    - The apply() function is used to join the lemmatized tokens (stored in tok_lem) into a single string for each tweet using the join() method. This results in a cleaned, tokenized string that can be used as input for the machine learning model.
2. **Feature and Label Extraction**:
    - The X variable contains the **final text** of the tweets, and y contains the corresponding sentiment labels (positive or negative).
    - X is extracted using df['final_text'].values, and y is extracted using df['label'].values.
3. **Verifying the Dataset Structure**:
    - len(X) and len(y) confirm the number of samples in the feature set and label set, ensuring they match.
    - pd.Series(y).value_counts() is used to display the distribution of sentiment labels, confirming that the dataset is balanced.

Title of Task 14: Train-Test Split

In this task, we split the dataset into **training** and **testing** sets. The training set is used to train the machine learning model, while the testing set is used to evaluate the model's performance. The split is done in a way that ensures the sentiment distribution is similar in both sets by using **stratified sampling**. This ensures that both positive and negative sentiments are represented proportionally in both the training and testing datasets.

Code for Task 14:

```
TEST_SIZE = 0.2
RANDOM_STATE = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=TEST_SIZE, random_state=RANDOM_STATE, stratify=y)
len(X_train), len(X_test)
```

Sample Output of Task 14:

```
(160000, 40000)
```

Code Description of Task 14:

1. **Train-Test Split**:
   - We use the train_test_split() function from **Scikit-learn** to divide the data into training and testing sets.
   - test_size=0.2 specifies that **20%** of the data will be used for testing and the remaining **80%** for training.
   - **Stratified Sampling** is used by setting stratify=y, ensuring that both the training and testing sets maintain the same distribution of sentiment labels as the original dataset.
2. **Displaying the Split Sizes**:
   - We print the lengths of X_train and X_test to confirm that the split has been applied correctly and the number of samples in each set is as expected.

Title of Task 15: Text Vectorization with TF-IDF

In this task, we transform the text data into numerical features using the **TF-IDF (Term Frequency-Inverse Document Frequency)** method. This is a common technique for converting text into vectors that machine learning models can understand. The **TF-IDF Vectorizer** converts each word in the text into a numerical value based on its frequency in the document and across the entire corpus. We also use **n-grams** (unigrams and bigrams) to capture word sequences, which can help understand context and improve model performance.

Code for Task 15:

```
MAX_FEATURES = 10000
NGRAMS = (1,2)
tfidf = TfidfVectorizer(max_features=MAX_FEATURES, ngram_range=NGRAMS)
X_train_vec = tfidf.fit_transform(X_train)
X_test_vec = tfidf.transform(X_test)
X_train_vec.shape, X_test_vec.shape
```

Sample Output of Task 15:

```
((160000, 10000), (40000, 10000))
```

Code Description of Task 15:

1. **TF-IDF Vectorizer Initialization**:
    - The TfidfVectorizer is initialized with two parameters:
        - max_features=10000: Limits the number of features (words or n-grams) to **10,000**. This helps reduce dimensionality and focuses on the most important words.
        - ngram_range=(1, 2): This specifies that both **unigrams** (single words) and **bigrams** (two consecutive words) will be used as features, capturing word sequences in addition to individual words.
2. **Fitting and Transforming the Training Data**:
    - The fit_transform() method is used to fit the vectorizer to the training data (X_train) and transform it into a matrix of numerical features.
3. **Transforming the Test Data**:
    - The transform() method is applied to the test data (X_test) using the previously fitted vectorizer. This ensures that the test data is represented using the same vocabulary learned from the training data.
4. **Displaying the Shape of the Transformed Data**:
    - We print the shapes of X_train_vec and X_test_vec to confirm that the text has been successfully converted into a matrix of features, with the number of rows representing the number of samples and the number of columns representing the features.

Title of Task 16: Naïve Bayes Model Training

In this task, we train a **Multinomial Naïve Bayes** classifier on the training data. The **Naïve Bayes** algorithm is commonly used for text classification tasks, such as sentiment analysis, because of its simplicity and effectiveness. The model is trained using the **TF-IDF vectorized** features, where the training set (X_train_vec) and the corresponding sentiment labels (y_train) are used. We specify the smoothing parameter alpha=1.0 to avoid zero probabilities and set fit_prior=True to allow the model to learn the class priors from the training data.

Code for Task 16:

```
clf = MultinomialNB(alpha=1.0,fit_prior =True,class_prior=None
                    )
clf.fit(X_train_vec, y_train)
print("all done ")
```

Sample Output of Task 16:

```
all done
```

Code Description of Task 16:

1. **Initializing the Classifier**:
   - We initialize the **Multinomial Naïve Bayes (MultinomialNB)** classifier with the following parameters:
     - alpha=1.0: This is the smoothing parameter, which helps avoid zero probabilities by adding a small constant to all counts.
     - fit_prior=True: This allows the model to learn the class priors (i.e., the distribution of positive vs. negative classes) from the training data.
     - class_prior=None: This means that the class priors will be learned from the data (the default behavior).
2. **Training the Model**:
   - The fit() function is used to train the model on the **TF-IDF vectorized training data** (X_train_vec) and the corresponding sentiment labels (y_train).
3. **Confirming Model Training**:
   - After training, we print "all done" to confirm that the model has been successfully trained and is ready for making predictions.

Title of Task 17: Model Evaluation with Accuracy and Classification Report

In this task, we evaluate the performance of the trained Naïve Bayes model. We start by making predictions on the test set **using the predict() method. The performance is then measured by calculating the** accuracy **of the model, which tells us the proportion of correctly classified instances. Additionally, we generate a** classification report**, which provides detailed metrics such as precision, recall, and F1-score for both positive and negative sentiments. These metrics help assess how well the model handles each sentiment class.**

Code for Task 17:

```
y_pred = clf.predict(X_test_vec)
acc = accuracy_score(y_test, y_pred)
print('Accuracy:', round(acc,4))
print(classification_report(y_test, y_pred, digits=4))
```

Sample Output of Task 17:

```
Accuracy: 0.7484
              precision    recall  f1-score   support

    negative     0.7445    0.7565    0.7505     20000
    positive     0.7525    0.7403    0.7464     20000

    accuracy                         0.7484     40000
   macro avg     0.7485    0.7484    0.7484     40000
weighted avg     0.7485    0.7484    0.7484     40000
```

Code Description of Task 17:

1. **Making Predictions**:
   - The predict() function is used to make predictions on the test set (X_test_vec). This step generates the predicted sentiment labels (y_pred), which are then compared with the actual labels (y_test).
2. **Calculating Accuracy**:
   - The accuracy_score() function from **Scikit-learn** is used to calculate the **accuracy** of the model, which is the proportion of correctly predicted labels. We round the accuracy to four decimal places using round() for clarity.
3. **Displaying the Classification Report**:
   - The classification_report() function provides precision, recall, F1-score, and support for each class (positive and negative). The digits=4 argument ensures that the metrics are displayed with four decimal places.
4. **Interpreting the Metrics**:
   - **Precision** tells us how many of the predicted positive or negative instances are actually correct.
   - **Recall** measures how many actual positive or negative instances were correctly identified.
   - **F1-score** is the harmonic mean of precision and recall, providing a single metric to evaluate the model's performance.

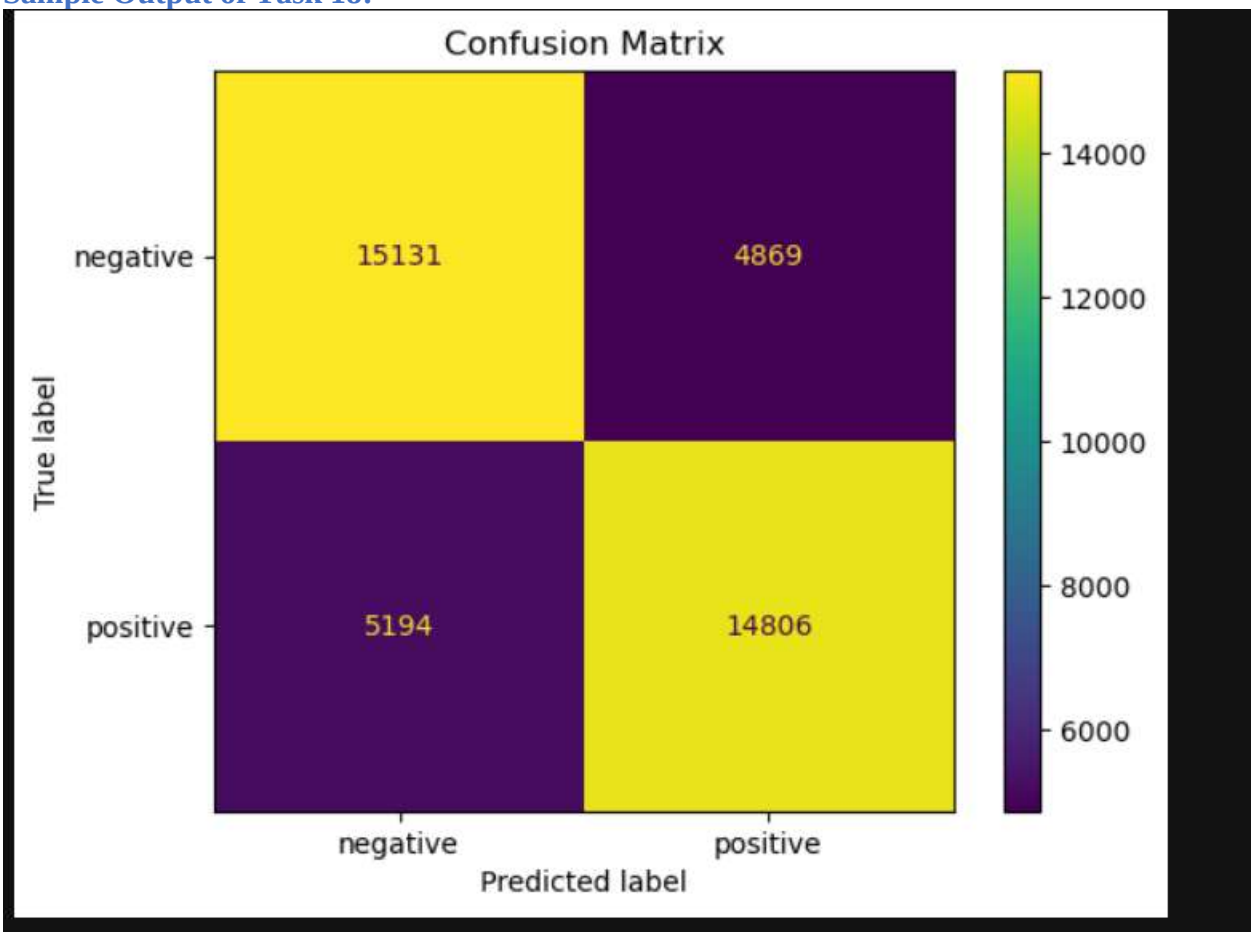
Title of Task 18: Confusion Matrix Visualization

In this task, we visualize the model's performance using a **confusion matrix**. A confusion matrix helps assess how well the model performs by comparing the actual and predicted labels. It provides a clear view of how many instances were correctly classified as positive or negative (true positives and true negatives) and how many were misclassified (false positives and false negatives). We visualize the matrix using a **heatmap** to make it easier to interpret.

Code for Task 18:

```
cm = confusion_matrix(y_test, y_pred, labels=np.unique(y_test))
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=np.unique(y_test))
fig, ax = plt.subplots()
disp.plot(ax=ax)
plt.title('Confusion Matrix')
plt.show()
```

Sample Output of Task 18:



Code Description of Task 18:

1. **Confusion Matrix Calculation**:
   - The confusion_matrix() function from **Scikit-learn** computes the confusion matrix by comparing the **true labels** (y_test) and the **predicted labels** (y_pred). The labels=np.unique(y_test) argument ensures that the matrix includes all possible labels in the dataset.
2. **Confusion Matrix Display**:
   - We create a **ConfusionMatrixDisplay** object using the calculated confusion matrix (cm) and the unique sentiment labels (np.unique(y_test)).
   - The plot() function is used to create the confusion matrix plot, which is displayed using plt.show().

3. **Plot Customization**:
    - We add a title (plt.title('Confusion Matrix')) to the plot to provide context for the visualization.

Title of Task 19: Model Prediction on New Examples

In this task, we test the trained model on new, unseen examples to make predictions. We use the TF-IDF vectorizer **to transform the new examples into the same feature space that was learned during training. Once the examples are vectorized, we use the trained** Naïve Bayes model **to predict the sentiment (positive or negative) for each example. This allows us to evaluate how well the model generalizes to new data.**

Code for Task 19:

```
examples = [
' good',
'I hate this ',

]
ex_vec = tfidf.transform(examples)
print(pd.DataFrame({'text': examples, 'pred': clf.predict(ex_vec)}))
```

Sample Output of Task 19:

```
           text      pred
 0         good  positive
 1  I hate this   negative
```

Code Description of Task 19:

1. **New Examples**:
    - We define a list of new examples (examples) for which we want to make predictions. These are simple sentences that were not part of the training data.
2. **Vectorizing the Examples**:
    - We use the transform() method of the trained **TF-IDF vectorizer** (tfidf) to convert the new text examples into feature vectors. These vectors represent the text in the same way the training data was represented.
3. **Making Predictions**:
    - The predict() method of the trained **Naïve Bayes classifier** (clf) is used to predict the sentiment for each transformed example. The model outputs a list of predictions, where each prediction corresponds to the sentiment (positive or negative).

4. **Displaying the Results**:
    - We use pd.DataFrame() to display the new examples along with their predicted sentiment in a readable format. The text column contains the original examples, and the pred column contains the predicted sentiment.

Project Code

```
import os, re, gc, string, random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, confusion_matrix,
ConfusionMatrixDisplay, accuracy_score
import nltk
from nltk.corpus import stopwords, wordnet
from nltk.stem import PorterStemmer, WordNetLemmatizer
from nltk.tokenize import word_tokenize
nltk.download('stopwords'); nltk.download('punkt'); nltk.download('wordnet');
nltk.download('omw-1.4')
import re
import string

columns = ["target", "id", "date", "flag", "user", "text"]
df = pd.read_csv("training.1600000.processed.noemoticon.csv", encoding='latin-
1', names=columns)


df = df[df['target'].isin([0, 4])]


sample_per_class = 100000

df = pd.concat([
   df[df['target']==0].sample(sample_per_class, random_state=42),
   df[df['target']==4].sample(sample_per_class, random_state=42)
])


df = df.reset_index(drop=True)
```

```
df.to_csv("balanced_sentiment140_no_neutral.csv", index=False)

print("new data set for  my project ")

label_map = {0:'negative',2:'neutral',4:'positive'}
df['label'] = df['target'].map(label_map).fillna(df['target'])
print(df[['text','label']].head())
print(df[['text','label']].tail())

counts = df['label'].value_counts()

plt.figure(figsize=(5,5))
plt.pie(
    counts,
    labels=counts.index,
    autopct='%1.1f%%',
    startangle=90
)
plt.title('Sentiment Distribution (Positive vs Negative)')
plt.show()


print("Null values per column:")
print(df.isnull().sum())


df = df.dropna(subset=['text', 'target'])

print("Dataset ok. final rows:", len(df))

df['text_lower'] = df['text'].str.lower()
print (df['text_lower'].head())

def clean_text(text):
    text = re.sub(r"http\S+", "", text)
    text = re.sub(r"@\w+", "", text)
    text = re.sub(r"#\w+", "", text)
    text = re.sub(r"[^a-zA-Z\s]", "", text)
    text = re.sub(r"\s+", " ", text).strip()
    return text

df["clean_text"] = df["text_lower"].apply(clean_text)
print(df[['clean_text']].head())
```

```
df['text_nopunct'] = df['clean_text'].apply(lambda x:
re.sub(f"[{re.escape(string.punctuation)}]", "", x))

print(df[['text_nopunct']].head())

df['tokens'] = df['text_nopunct'].apply(word_tokenize)
print(df[['tokens']].head())


stop_words = set(stopwords.words('english'))
df['tokens_nostop'] = df['tokens'].apply(lambda tokens: [word for word in tokens
if word not in stop_words])
print(df[['tokens_nostop']].head())

def synonym_sub(tokens):
   out = []
   for w in tokens:
      syns = wordnet.synsets(w)
      if syns:
         lemmas = syns[0].lemmas()
         if lemmas:
            out.append(lemmas[0].name().replace('_',' '))
         else:
            out.append(w)
      else:
         out.append(w)
   return out

df['tok_syn'] = df['tokens_nostop'].apply(synonym_sub)
print(df[['tok_syn']].head())

stemmer = PorterStemmer()
df['tok_stem'] = df['tok_syn'].apply(lambda toks: [stemmer.stem(w) for w in
toks])
df[['tok_stem']].head()


lemm = WordNetLemmatizer()
def lemmatize_tokens(toks):
   out = []
   for w in toks:
      l = lemm.lemmatize(w, pos='n')
      l = lemm.lemmatize(l, pos='v')
      l = lemm.lemmatize(l, pos='a')
      out.append(l)
   return out
```

```
df['tok_lem'] = df['tok_stem'].apply(lemmatize_tokens)
df[['tok_lem']].head()


df['final_text'] = df['tok_lem'].apply(lambda toks: ' '.join(toks))
X = df['final_text'].values
y = df['label'].values
len(X), len(y), pd.Series(y).value_counts()

TEST_SIZE = 0.2
RANDOM_STATE = 42
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=TEST_SIZE,
random_state=RANDOM_STATE, stratify=y)
len(X_train), len(X_test)

MAX_FEATURES = 10000
NGRAMS = (1,2)
tfidf = TfidfVectorizer(max_features=MAX_FEATURES,
ngram_range=NGRAMS)
X_train_vec = tfidf.fit_transform(X_train)
X_test_vec = tfidf.transform(X_test)
X_train_vec.shape, X_test_vec.shape

clf = MultinomialNB(alpha=1.0,fit_prior =True,class_prior=None
            )
clf.fit(X_train_vec, y_train)
print("all done ")

y_pred = clf.predict(X_test_vec)
acc = accuracy_score(y_test, y_pred)
print('Accuracy:', round(acc,4))
print(classification_report(y_test, y_pred, digits=4))

cm = confusion_matrix(y_test, y_pred, labels=np.unique(y_test))
disp = ConfusionMatrixDisplay(confusion_matrix=cm,
display_labels=np.unique(y_test))
fig, ax = plt.subplots()
disp.plot(ax=ax)
plt.title('Confusion Matrix')
plt.show()


examples = [
' good',
'I hate this ',
```

```
]
ex_vec = tfidf.transform(examples)
print(pd.DataFrame({'text': examples, 'pred': clf.predict(ex_vec)}))
```