The IMDB Dataset: Sentiment Analysis with a Support Vector Machine

1. Importing necessary libraries

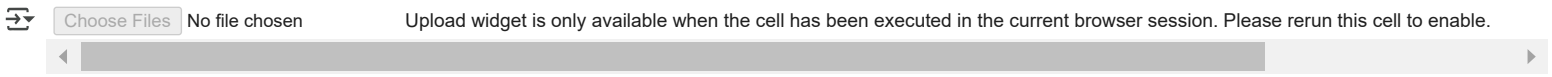
First, we need to import the libraries required for our analysis: `pandas` for data manipulation, `seaborn` and `matplotlib` for data visualisation, `nltk` (Natural Language Toolkit) for natural language processing tasks, `string` for string manipulation, and `sklearn` for building the support vector machine (SVM) and evaluating its performance.

```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import nltk
from nltk.corpus import stopwords
import string
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
```

2. Loading the dataset

Here, we use the IMDB Dataset from Kaggle. This dataset contains 50,000 movie reviews labeled as either positive or negative.

```
from google.colab import files
uploaded = files.upload()
```



The dataset is limited to the first 500 samples to reduce computational load and expedite processing.

```
data = pd.read_csv('IMDB Dataset.csv')
data = data.head(500)
```

3. Checking for missing values

Ensuring that there are no missing values is critical for the integrity of our analysis process. The `isnull().sum()` method counts the number of missing entries in each column. If any entries are missing in the `sentiment` or `review` columns, those rows are removed with `dropna()`.

```
print(data.isnull().sum())
```

```
review       0
sentiment    0
dtype: int64
```

```
data = data.dropna(subset=['sentiment', 'review'])
```

4. Preprocessing the text

In this section, we employ the NLTK library to download and define a set of stopwords in English. Stopwords are commonly used words (such as *and*, *the*, *is*, etc.) that are often removed during text preprocessing because they typically do not contribute meaningful information for sentiment classification. We also define a function, `preprocess_text`, to clean the text data by lowercasing and removing punctuations and stopwords.

```
nltk.download('stopwords')
stop_words = set(stopwords.words('english'))

def preprocess_text(text):
  # Converting to lowercase
  text = text.lower()
  # Removing punctuation
  text = text.translate(str.maketrans('', '', string.punctuation))
  # Removing stopwords
  text = ' '.join([word for word in text.split() if word not in stop_words])
  return text

data['cleaned_text'] = data['review'].apply(preprocess_text)
```

```
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
```

5. Mapping sentiment labels

The sentiment labels are mapped to numerical values: `positive` is converted to 1 and `negative` to 0. This transformation is necessary for the SVM, which operates on numerical data.

```
data['sentiment'] = data['sentiment'].map({'positive': 1, 'negative': 0})
```

6. Splitting the data into training and testing sets

We split the dataset into training and testing sets using the `train_test_split` function. We specify `test_size=0.2` to reserve 20% of the data for testing, and `random_state=42` to ensure the data is split consistently every time we run the code.

```
X_train, X_test, y_train, y_test = train_test_split(data['cleaned_text'], data['sentiment'],
                                                    test_size=0.2, random_state=42)
```

7. TF-IDF Vectorisation

Next, the TF-IDF (Term Frequency-Inverse Document Frequency) vectoriser is employed. It converts the cleaned text data into a matrix of TF-IDF features. TF-IDF is effective for capturing the importance of words in documents relative to a corpus, making it suitable for sentiment analysis.

```
vectorizer = TfidfVectorizer()
X_train_tfidf = vectorizer.fit_transform(X_train)
X_test_tfidf = vectorizer.transform(X_test)
```

8. Creating and training the model

Now, we create an instance of the SVM with a linear kernel, and train it. The model is trained on the TF-IDF transformed training data (`X_train_tfidf`) and the corresponding sentiment labels (`y_train`).

```
model = SVC(kernel='linear')
model.fit(X_train_tfidf, y_train)
```

```
SVC
SVC(kernel='linear')
```

9. Making predictions

Once the model is trained, we use it to make predictions on the test data (`X_test_tfidf`). The model generates predicted sentiment labels (`y_pred`) for the test set, which we will compare with the actual labels (`y_test`) to evaluate its performance.

```
y_pred = model.predict(X_test_tfidf)
```

10. Evaluating the model

Now, we calculate the accuracy of the model and print a detailed classification report, which includes precision, recall, and F1-score for each class, providing insight into the model's performance.

```
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')
print("\nClassification Report:\n", classification_report(y_test, y_pred, target_names=['negative', 'positive']))
```

```
Accuracy: 0.76

Classification Report:
              precision    recall  f1-score   support

    negative       0.69      0.87      0.77        46
    positive       0.86      0.67      0.75        54

    accuracy                           0.76       100
   macro avg       0.77      0.77      0.76       100
weighted avg       0.78      0.76      0.76       100
```

11. Generating the confusion matrix

Finally, we generate a confusion matrix to visualise the performance of the model. The confusion matrix displays the true positive, true negative, false positive, and false negative counts, allowing for an intuitive understanding of where the model is making errors.

```
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(4,4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=['negative', 'positive'],
            yticklabels=['negative', 'positive'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix Heatmap')
plt.show()
```