Coursework 1: Sentiment analysis from tweets (Report)

Code Explanation:

Question 1:

Data Loading: Data is retrieved from a tab-separated file using the **load_data** function. Every data line is appended to the **raw_data** list, and the header row is omitted. The **raw_data** list is converted to a tuple as it specifies in the question to return a tuple of **label** and **text**. The label and text from each line are extracted using the **parse_data_line** function.

Preprocessing and Data Splitting: The data is divided into training and testing sets according to a predetermined proportion using the **split_and_preprocess_data** function. To tokenize and preprocess the text data, it makes use of the **pre_process** function. The **to_feature_vector** function is then used to transform the parsed text into feature vectors. This function will be coded in Question 2.

parse_data_line() : This function returns the label and text from the data.

pre_process() : This function uses regular expression to find all the alphanumeric characters in the text. The function **re.findall**(r'\w+', text) locates every word character sequence, letters, numbers, or underscores in the given sentence. Every word from the original sentence will be included in the tokens list that is produced.

Question 2:

to_feature_vector() : A dictionary-based feature vector is created from a list of tokens using the **to_feature_vector** function. The strategy used is called Binary Feature Values: Every token is expressed as a binary feature, where a token's presence is indicated by the number 1 and its absence by the number 0. By using this method, the feature vector is reduced to a binary representation.

Global Feature Dictionary: To track the occurrences of each feature throughout the dataset, the code keeps track of them in a global dictionary called **global_feature_dict**. In this dictionary, the count is increased with each token that is encountered. This global dictionary is a useful tool for comprehending how features are distributed over the whole dataset.

train_classifier() : Within an NLTK wrapper, the **train_classifier** method uses LinearSVC from scikit-learn as the classifier. The Pipeline class is used to design a machine learning pipeline, of which the support vector machine is the only component. The classifier is then trained using the training data.

Question 3:

cross_validate() : The code uses KFold for cross-validation and scikit-learn metrics. By doing k-fold cross-validation, the **cross_validate** function assesses the classifier's performance on various dataset subsets. The function initialises the accuracy, precision, recall, and F1-score in a dictionary called results. Folds are created from the dataset, and for each fold: The training fold is used to train the classifier. The test fold is used to make predictions. The results are updated and metrics for the fold are computed. The average scores are calculated and given back for each fold.

Predict_labels() and **predict_label_from_raw() :** Using the **classify_many** function of the trained classifier, the **predict_labels** function forecasts labels for preprocessed samples. Using the pipeline previously established, the **predict_label_from_raw** function first preprocesses raw text into a feature vector. The trained classifier is then used to predict the label. Predictions can be made with flexibility using these functions on both raw and pre-processed text data.

# MAIN: The data is loaded from the designated dataset file and global data lists are initialised in the main section. In order to prepare the data for cross-validation and prediction, it then divides the

dataset into training and test sets. Then the **cross_validate** function is called to check the classifiers performance on the train data.

Question 4:

confusion_matrix_heatmap() : The **confusion_matrix_heatmap** function uses matplotlib and scikit-learn metrics to create a heatmap that visually displays a classifier's performance.

error_analysis() and print_error_to_file() : False positives and false negatives are detected by error analysis functions and are stored in "false_positives.txt" and "false_negatives.txt." The predicted labels (predicted_labels) and true labels (true_labels) for the test set are obtained after the classifier is trained on the train_data in the main section.
A classifier is trained using the supplied training data (train_data) in the main section. Next, we obtain the predicted labels (predicted_labels) and true labels (true_labels) for the test set. To find and print false positives and false negatives to files, the error_analysis function is called. Lastly, the **confusion_matrix_heatmap** function is used to plot the confusion matrix heatmap.

Observations on the error types:

False_positive : (Example for **Instance 3** in the false_positives.txt file)The text talks about protestors in Jharkhand who are calling for the release of a tribal leader. It's possible that the classifier misread the context, producing a false positive. There are a total of **589** false positive values in my code.

False_negative : (Example for **Instance 1** in the false_negatives.txt file) In the news article about medical marijuana and epilepsy, a neurologist is mentioned in the text. There could have been a false negative because the classifier disregarded negative sentiment cues. There are a total of **447** false negative values in my code.

Question 5:

Iteration 1: Several changes were made to the **`to_feature_vector`** function in the fifth section of the code to improve feature representation. **`word_tokenize`** from NLTK is now used in the tokenization process, and **`WordNetLemmatizer`** lemmatization has been added for normalisation. Furthermore, the feature set is made clearer and more significant by eliminating stopwords and non-alphanumeric tokens. The addition of unigram tokens, bigrams, and trigrams enriches the feature representation. The goal of this varied feature set is to extract more complex patterns and semantic subtleties from the textual data. All of these modifications raise the level of complexity in the feature extraction procedure, which may result in better model performance on text classification tasks.

Iteration 2: To improve the feature extraction procedure, the functions **words_per_sentence(text)** and **pre_process(text)** are added to the modified code. In addition to providing additional structural information, the **words_per_sentence(text)** function counts the words in each sentence in the input text. After that, this data is included in the preprocessing and tokenization stages of the **pre_process(text)** function. The count of words per sentence is added to the list of tokens by appending the calculated **words_per_sentence_list**, which enhances the feature set. The goal of this change is to record more subtle patterns pertaining to sentence structure. Furthermore, more features are added to the feature vector in the **to_feature_vector function**. The number of words in each sentence is added as a feature, along with unigram tokens, bigrams, and character n-grams, to create a more thorough representation of the input data.

Classifier	**Precision**	**Recall**	**F1 Score**	**Accuracy**
SVM Question 1 to 4	0.836	0.837	0.836	0.837
Iteration 1 Question 5	0.844	0.846	0.845	0.846
Iteration 2 Question 5	0.844	0.846	0.845	0.865

Table 1: Result Table