Student Name: Syam Immanuel Paul Bondada

Roll Number: 230853737

Question 1: Input and Basic preprocessing (10 marks):

The parse_data_line function extracts label and text information from a data line, preparing it for further processing. It uses regular expressions to separate punctuation at the beginning and end of words, ensuring punctuation marks are treated as separate tokens during tokenization. The function returns a tuple containing the label and text.

The pre_process function further prepares text for analysis. It utilizes regular expressions to separate punctuation, tokenizes the text using whitespace, and normalizes tokens to lowercase. The result is a list of pre-processed tokens ready for natural language processing tasks.

Question 2: Basic Feature Extraction (20 marks):

The function initializes an empty dictionary called feature_vector that will store the features (tokens) and their corresponding weights (occurrence counts). It then iterates through each token in the input list of tokens. For each token, it updates the feature_vector by counting the occurrences of that token. The get method is used to retrieve the current count of the token. If the token is not in the dictionary, it defaults to 0, and 1 is added to it.

Additionally, the function updates a global feature dictionary (global_feature_dict) with the counts of each token across all feature vectors. This dictionary is global, meaning it accumulates counts across different calls to the to_feature_vector function.

Question 3: Cross-validation (20 marks):

The cross_validate function is designed to perform cross-validation on a given dataset, training and evaluating a classifier across multiple folds. Here's a summarized breakdown of its functionality:

1. Input Parameters:
   - dataset: A list of samples, where each sample is a tuple containing features and labels.
   - folds: The number of folds for cross-validation, determining the subsets for training and testing.
2. Initialization:
   - results: A dictionary to store performance metrics (precision, recall, F1-score, and accuracy) for each fold.
   - fold_size: The size of each fold, calculated based on the length of the dataset and the specified number of folds.
3. Cross-Validation Loop:
   - Iterates through each fold, selecting different subsets for training and testing in each iteration.
4. Train-Test Split:
   - Splits the dataset into training and testing folds for each iteration.
     - test_data_fold: The subset of the dataset used for testing the classifier.
     - train_data_fold: The remaining data used for training the classifier.
5. Classifier Training:
   - Trains the classifier using the train_classifier function on the training fold.
6. Classifier Testing:
   - Uses the trained classifier to predict labels on the testing fold (test_samples).
   - Extracts the true labels (true_labels) from the testing fold.
7. Classifier Evaluation:
   - Evaluates the classifier's performance using the classification_report function, producing precision, recall, F1-score, and accuracy.
   - Stores the results in the fold_results dictionary.
8. Results Storage and Printing:

    - Appends the fold_results dictionary, containing performance metrics for the current fold, to the results dictionary.
    - Prints the performance metrics for each fold.
9. Average Performance Calculation:
    - After processing all folds, calculates the average performance metrics across all folds.
    - Stores the average metrics in the avg_results dictionary.

The function orchestrates the process of training and evaluating a classifier multiple times on different subsets of the dataset, providing a comprehensive view of the model's performance through cross-validation.

Question 4: Error Analysis (20 marks)

confusion_matrix_heatmap Function:

The confusion_matrix_heatmap function is designed to create a visually appealing and informative heatmap representation of a confusion matrix, a common evaluation metric in classification problems. Here's a detailed report on what this function does:

Confusion Matrix Calculation:

The metrics.confusion_matrix function from scikit-learn is used to calculate the confusion matrix based on the true labels (y_test) and predicted labels (preds).
The confusion matrix represents the counts of true positive, true negative, false positive, and false negative predictions.

Plotting the Heatmap:

A matplotlib figure is created with a specified size (figsize).
The confusion matrix is visualized as a heatmap using the matshow function, with the color intensity representing the count of each class combination.

cross_validate_with_error_analysis Function:

The cross_validate_with_error_analysis function combines the training, testing, and evaluation of a classifier in a k-fold cross-validation setup. Additionally, it performs error analysis by displaying a confusion matrix heatmap and printing false positives and false negatives for the positive label. Here's a detailed report on what this function does:

Error Analysis - False Positives and False Negatives:

False positives and false negatives for the positive label are identified and printed for each fold.

The confusion_matrix_heatmap function provides a clear visualization of the classifier's performance through a heatmap representation of the confusion matrix. The cross_validate_with_error_analysis function extends the analysis by incorporating k-fold cross-validation, providing a more robust assessment of the classifier's generalization performance and including error analysis for the positive label. Together, these functions support comprehensive evaluation and diagnostic analysis of a text classification model.

Questions 5: Optimising pre-processing and feature extraction (30 marks)
Improvements In pre-processing:

1. Tokenization: Here in the improved version I used re.split(r"\s+", text) for tokenization, which is a simpler and more straightforward approach than word_tokenize from NLTK. It splits the text based on whitespace.

2. Custom Stopwords: The custom_stopwords set consists of user-defined stopwords, which, in this example, includes common words such as "list," "of," "custom," and "stopwords." The purpose of this set is to identify and exclude these specific words during the text pre-processing stage.

3. Punctuation Handling: These punctuation handling steps contribute to the overall preprocessing of the text data. Separating punctuation from words helps ensure that each token (word) is treated as an individual entity during subsequent processing steps, such as lowercase conversion and lemmatization. This can be beneficial for sentiment analysis tasks as it allows the model to focus on the semantic content of words while disregarding attached punctuation marks.

4.Normalization:

Lowercasing: The line tokens = [t.lower() for t in tokens if t.lower() not in custom_stopwords] ensures that all words in the text are converted to lowercase. Lowercasing helps standardize the text by treating uppercase and lowercase versions of the same word as identical. This is essential for consistency in subsequent analyses.

Lemmatization: The lemmatization step is performed using the WordNetLemmatizer from the NLTK library: tokens = [lemmatizer.lemmatize(t) for t in tokens]. Lemmatization involves reducing words to their base or root form. For example, lemmatizing "running" would result in "run." This step aims to ensure that different inflections or derivations of a word are treated as the same, reducing the dimensionality of the feature space.

Lexicon-based features:

Positive words: The presence of positive words in the text suggests a positive sentiment. The more positive words there are, the more likely the text is to express a positive sentiment.

Negative words: The presence of negative words in the text suggests a negative sentiment. The more negative words there are, the more likely the text is to express a negative sentiment.

By combining information about positive and negative words, lexicon-based features can provide a more accurate assessment of the sentiment expressed in the text. When combined with normal feature extraction techniques, these features can help to improve the overall accuracy of sentiment analysis models.

Stylistic features:

Average number of words per sentence: A higher average number of words per sentence suggests a more formal or analytical style of writing. This style is often associated with more neutral or objective sentiment.

Use of exclamation points: The use of exclamation points suggests a more emotional or emphatic style of writing. This style is often associated with more positive or negative sentiment.
Grid search and parameter optimization techniques like it are crucial for improving the accuracy of machine learning models, including support vector machines (SVMs). Compared to using a fixed or default parameter value, grid search helps to identify the optimal parameter settings that lead to the best performance on a given dataset.

The second approach, which involves cross-validation, demonstrates a more stable and higher overall performance, as indicated by the consistent accuracy and elevated F1-score. This suggests that the model trained in the second case generalizes well across different subsets of the data, providing more reliable and robust predictions compared to the individual fold-wise evaluations in the first case.