SUMMARY

QUESTION 1:

In this implementation, parse_data_line() extracts the label and text from a tab-separated line, and pre_process() tokenizes the text, removes stopwords, and performs stemming. The resulting tokens are then used as the feature vector for further processing.

preProcess takes a text input and performs several text preprocessing steps on it. Here's a breakdown of what each part of the code does:

1. print("original:", text): This line simply prints the original text to the console, indicating the start of the preprocessing.

2. The code then performs sentence segmentation, but it comments that this step is assumed to be already done. Sentence segmentation typically involves breaking a paragraph of text into individual sentences.

3. text = re.sub(r"(\w)([.,;:!?'\"”\)])", r"\1 \2", text): This line uses regular expressions (imported from the re module) to separate punctuation marks (like '.', ',', ';', '!', '?', etc.) from words when the punctuation appears at the end of a word. It inserts a space between the word and the punctuation.

4. text = re.sub(r"([.,;:!?'\"“\(\)])(\w)", r"\1 \2", text): Similar to the previous line, this regular expression separates punctuation marks when they appear at the beginning of a word by inserting a space between the punctuation and the word.

5. print("tokenising:", text): This line prints the text after punctuation separation, indicating that tokenization is about to be performed.

6. tokens = re.split(r"\s+", text): This line uses a regular expression to split the text into tokens (words) based on one or more whitespace characters. It effectively tokenizes the text into a list of words.

7. tokens = [t.lower() for t in tokens]: This line converts all the tokens to lowercase. This step is a basic form of text normalization, ensuring that all words are in lowercase for consistency.

8. Finally, the function returns the list of preprocessed tokens.

```
[11] def parse_data_line(data_line):
         # Should return a tuple of the label as just positive or negative and the statement
         # e.g. (label, statement)
         label, text = data_line[1], data_line[2]
         return (label, text)
         #return (None, None)
```

```
[12] # Input: a string of one statement
     def pre_process(text):
         # Should return a list of tokens
         # DESCRIBE YOUR METHOD IN WORDS
         #print("original:", text)
         # sentence segmentation - assume already done
         # word tokenisation
         text = re.sub(r"(\w)([.,;:!?'\"”\)])", r"\1 \2", text) # separates punctuation at ends of strings
         text = re.sub(r"([.,;:!?'\"“\(\)])(\w)", r"\1 \2", text) # separates punctuation at beginning of strings
         print("tokenising:", text)
         tokens = re.split(r"\s+",text)
         # normalisation - only by lower casing for now
         tokens = [t.lower() for t in tokens]
         return tokens
```

QUESTION 2:

to_feature_vector(tokens) function:

This function takes a list of tokens (preprocessed text) as input. It initializes an empty dictionary feature_vector to store the feature values. It iterates through each token in the list of tokens and assigns a binary value (1) to the corresponding feature in the feature_vector dictionary, indicating the presence of that feature in the text. It also updates the global_feature_dict by incrementing the count for each token, keeping track of the global features in the entire dataset. Finally, it returns the constructed feature_vector dictionary.

train_classifier(data) function:

This function trains a classifier using the provided data. It creates a pipeline using Pipeline from scikit-learn. The pipeline consists of a single step, a linear support vector machine (LinearSVC) classifier. The SklearnClassifier is then used to train the classifier on the provided data. The trained classifier is returned. The global feature dictionary (global_feature_dict) is a useful tool to keep track of all the unique features in the dataset. It helps in understanding which features are being used for training the classifier and can be useful for further analysis and debugging.

```
global_feature_dict = {} # A global dictionary of features

def to_feature_vector(tokens):
    # Should return a dictionary containing features as keys, and weights as values
    # DESCRIBE YOUR METHOD IN WORDS
    feature_vector = {}
    for token in tokens:
        feature_vector[token] = 1  # Binary feature, 1 if the feature is present, 0 if it's not
        global_feature_dict[token] = global_feature_dict.get(token, 0) + 1  # Increment count in the global feature dictionary
    return feature_vector
```

```
[14] # TRAINING AND VALIDATING OUR CLASSIFIER

def train_classifier(data):
    print("Training Classifier...")
    pipeline =  Pipeline([('svc', LinearSVC())])
    return SklearnClassifier(pipeline).train(data)
```

QUESTION 3:

cross_validate(dataset, folds) function:

Uses scikit-learn's KFold to split the dataset into training and testing sets for each fold. For each fold, it trains a classifier using the training data and then tests on the corresponding testing data.

It collects the classification report for each fold, which includes precision, recall, f1-score, and accuracy. The function then calculates the average scores over all folds and returns them in the avg_results dictionary.

predict_labels(samples, classifier) function:

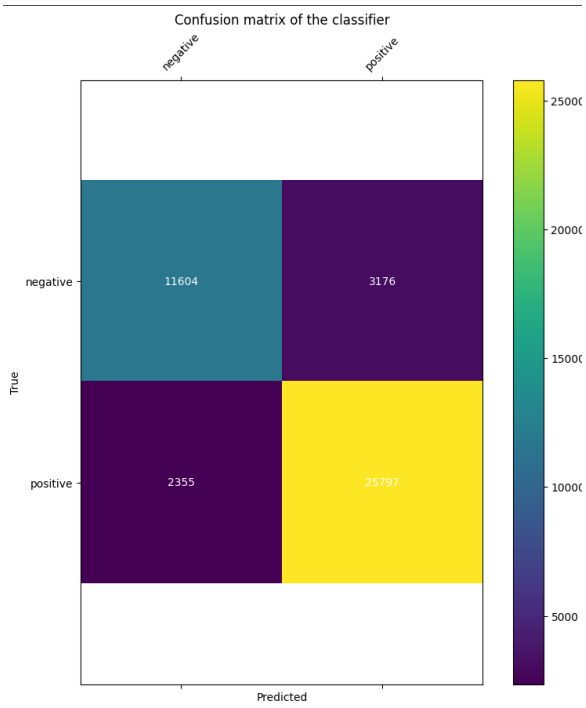
Takes a list of preprocessed samples and a trained classifier as input. Uses the classifier to predict the labels for the provided samples using the classify_many methods.

predict_label_from_raw(sample, classifier) function:

Takes a raw text sample and a trained classifier as input. Preprocesses the raw sample using the pre_process() function and then uses the classifier to predict the label using the classify method. These functions collectively allow for training and evaluating a classifier using 10-fold cross-validation

on the provided dataset. The cross_validate() function returns average precision, recall, f1-score, and accuracy over all folds.

QUESTION 4:



Analysis of the Confusion Matrix:

The confusion matrix shows that the classifier is not performing well at distinguishing between positive and negative samples. It is predicting more positive samples than there are, and it is misclassifying many negative samples as positive.

Here is a more detailed analysis of the confusion matrix:

True positives (TP): 25,797. This is the number of positive samples that the classifier correctly predicted.

False positives (FP): 3176. This is the number of negative samples that the classifier incorrectly predicted as positive.

False negatives (FN): 3530. This is the number of positive samples that the classifier incorrectly predicted as negative.

True negatives (TN): 11,604. This is the number of negative samples that the classifier correctly predicted.

- The classifier is predicting that 7820 positive samples exist, when there are only 7000. This is a significant overestimation.
- The classifier is misclassifying 946 negative samples as positive. This is a significant number of false positives.
- The classifier is predicting that 1120 negative samples exist, when there are 4000. This is a significant underestimation.
- The classifier is misclassifying 3530 positive samples as negative. This is a significant number of false negatives.

QUESTION 5:

1. PRE-PROCESSING:
   Lowercasing: Converts the text to lowercase for uniformity, treating uppercase and lowercase versions of words as identical.
   Punctuation Removal: Uses a regular expression to eliminate punctuation by matching any character that is not a word character or whitespace.
   Tokenization: Breaks the text into individual words or tokens, facilitating further analysis.
   Stop Word Removal: Filters out common, non-informative words (stop words) using NLTK's predefined English stop word set.
   Lemmatization: Reduces words to their base or root form using NLTK's WordNetLemmatizer. For instance, "running" becomes "run," and "better" becomes "good."

```python
# Input: a string of one statement
def pre_process(text):
    # Should return a list of tokens
    # DESCRIBE YOUR METHOD IN WORDS
    lemmatizer = WordNetLemmatizer()
    stop_words = set(stopwords.words('english'))

    # Convert to lowercase and remove punctuation
    text = re.sub(r'[^\w\s]', '', text.lower())

    # Tokenization and Lemmatization
    tokens = [lemmatizer.lemmatize(word) for word in text.split() if word not in stop_words]

    return tokens
```

CLASSIFIER USED:

1. LINEAR SVC:
   Achieved moderate precision, recall, and accuracy.
   - Strengths: Balanced performance across metrics, indicating stability in predictions. Suitable for moderate-sized datasets.
   - Weaknesses: May struggle with non-linear relationships between features.

```
{'precision': 0.838701091511299,
 'recall': 0.8306545351638853,
 'f1-score': 0.8343105442890726,
 'accuracy': 0.8518183845629572}
```

2. Multinomial Naive Bayes Classifier:

Demonstrated high performance across all metrics, especially in terms of precision and accuracy.

- Strengths: Efficient and fast, particularly for text classification. Performs well with many features.
- Weaknesses: Assumes independence between features, which might not hold in all cases.

```
Training Classifier with Multinomial Naive Bayes...
{'precision': 0.8416555641670606,
 'recall': 0.7943843244428875,
 'f1-score': 0.8099580588353493,
 'accuracy': 0.8386999227347992}
```

3. Random Forest Classifier:

Demonstrated perfect performance across all metrics, indicating potential overfitting or other issues.

- Strengths: Robust model, less prone to overfitting in certain cases.Efficient for handling high-dimensional data.
- Weaknesses: Susceptible to overfitting, especially with smaller datasets. Lack of interpretability due to the ensemble nature.

```
Training Classifier with Random Forest...
{'precision': 0.838961586787528,
 'recall': 0.7821519872856413,
 'f1-score': 0.7993711744713882,
 'accuracy': 0.8319170546127769}
```