IRTM Homework3 Report

B10705054 劉知微

1 Execution Environment

The code is executed in vscode python.

2 Programming Language

The programming language used is Python 3.

3 Execution Method

To run the code in VS Code, you may need to install additional libraries and set up the required environment. Ensure that all necessary packages, such as `numpy` and `nltk`, are installed before execution. The following Python modules are used in the code:

```
from collections import defaultdict
import os
from nltk.stem import PorterStemmer
import re
import math
import csv
```

Please ensure that the `data` folder and the `stopwords.txt` file (which is included in the folder) and `training.txt` are prepared in the same directory as the code. The folder and stopwords file paths should be defined as follows:

```
document_folder = "./data"
stopword_file = './stopwords.txt'
training_file = './training.txt'
```

Lastly, to execute the code in VS Code, you can either click the "Run" button in the upper-right corner or type `python pa3.py` in the terminal. This will execute the code in the active code.

4 Workflow

4.1 Tokenization and split training data

To tokenize the text, the preprocessing involves several steps. First, regular expressions are used to remove punctuation marks and digits from the input text. The text is then converted to lowercase to ensure consistency across all words. Next, the `split()` function is applied to break the text into individual words, and any extra spaces around the words are removed.

After tokenization, Porter's stemming algorithm is applied to reduce the words to their root forms. Finally, stopwords are filtered out, leaving only the relevant tokens in the list `token_list`, which is returned as the result.

```
1  def tokenization(text):
2      doc = re.sub(r"[^\w\s]|[\d]", " ", text)
3      doc = doc.lower()
4
5      words = [word.strip() for word in doc.split() if word.strip()]
6
7      # Porter's stemming
8      stemmer = PorterStemmer()
9      stemming = [stemmer.stem(word) for word in words]
10
11     # Stop words removal
12     token_list = [word for word in stemming if word not in
           stop_words]
13
14     return token_list
```

I read the files from the folder `data`. Each file path is generated using `os.path.join()`, and the file is opened for reading. Every file goes through the previously mentioned `tokenization()` function, and the tokenized content is appended to the list `documents`. Later on i read the training data from training.text. After that, the load_training_data function creates the class_to_docs dictionary, which maps class IDs to corresponding document IDs. The separate_train_test function then takes this class_to_docs dic-

tionary and a list of documents (documents). It first determines the set of training document IDs by extracting the document IDs from class_to_docs. The function assumes that document IDs range from 1 to N, where N is the total number of documents. It then calculates the test document IDs as those that are not included in the training set.

Next, the function organizes the training documents by class, creating a dictionary (class_documents) where each class ID maps to a list of tuples. Each tuple contains a document ID and its corresponding content. Similarly, the test documents are collected into a list of tuples containing document IDs and their content.

Finally, the function returns two collections: class_documents, which contains the training data organized by class, and test_documents, which contains the test data.

4.2 Training

The training process involves several key steps to prepare the data for classification using the Multinomial Naive Bayes algorithm. The following functions are called in sequence during training:

1. **Extract Vocabulary**: The training process begins by extracting the vocabulary from the training documents using the `extract_vocabulary` function. This function collects all unique terms from the documents in `class_documents` and stores them in a set called `vocab`.

2. **Count Documents**: The `count_docs` function calculates the total number of documents in the training set by summing the lengths of the document lists in `class_documents`.

3. **Calculate Prior Probabilities**: The `calculate_prior` function calculates the prior probabilities for each class. The prior for each class is the fraction of documents in that class relative to the total number of documents. These are stored in a dictionary `prior`, where each class ID maps to its prior probability.

4. **Count Term Documents**: The `count_term_documents` function computes how many documents each term appears in across the classes. The result is a dictionary `term_doc_counts`, which stores the document frequency for each term in each class.

5. **Chi-Squared Feature Selection**: The `chi2_feature_selection` function applies the chi-squared test to select the most informative features (terms). It calculates the chi-squared statistic for each term and selects the top `num_features`

based on their chi-squared scores. This helps reduce the dimensionality of the model and focuses on the most relevant terms.

```
def chi2_feature_selection(class_documents, vocab,
    term_doc_counts, total_docs, num_features):
chi2_scores = defaultdict(float)
N = total_docs
for term in vocab:

    for class_id, doc_ids in class_documents.items():
        # True Positive (A): term appears in class
        A = term_doc_counts[class_id].get(term, 0)
        # False Positive (B): term appears in other classes
        B = sum( term_doc_counts[other_class].get(term, 0)
              for other_class in class_documents.keys() if
                 other_class != class_id)
        # False Negative (C): term doesn't appear in class
        C = len(doc_ids) - A
        # True Negative (D): term doesn't appear in other
            classes
        D = N - (A + B + C)

        # Calculate expected frequencies
        E_11 = (A + B) * (A + C) / N
        E_10 = (A + B) * (B + D) / N
        E_01 = (C + D) * (A + C) / N
        E_00 = (C + D) * (B + D) / N

        # Calculate chi-squared for each cell
        chi2 = 0
        for obs, exp in [(A, E_11), (B, E_10), (C, E_01), (D,
            E_00)]:
            if exp > 0:
                chi2 += (obs - exp) ** 2 / exp
        chi2_scores[term] += chi2

# Sort terms by chi-squared score
sorted_terms = sorted(chi2_scores.items(), key=lambda x:
    x[1], reverse=True)

```

```
    # Select top terms based on num_features
    top_terms = set(term for term, value in
        sorted_terms[:num_features])
    return top_terms
```

6. **Generate Term Counts**: The `generate_term_counts` function generates a frequency count of selected terms for each class in the training set. It creates a dictionary `term_counts` where each class ID maps to a dictionary of term frequencies.

7. **Calculate Conditional Probabilities**: The `calculate_conditional_probabilities` function computes the conditional probabilities for each term given each class. These are stored in `cond_prob` using Laplace smoothing to account for terms not appearing in some classes.

4.3 Testing

The testing process involves applying the trained Multinomial Naive Bayes model to classify new, unseen documents. The following functions are used during the testing phase:

Apply Multinomial Naive Bayes: The `apply_multinomial_nb` function is used to classify test documents based on the learned prior and conditional probabilities. It first extracts the tokens from each test document and then computes the scores for each class by summing the logarithms of the prior and conditional probabilities for each term in the document. The class with the highest score is chosen as the predicted class.

```
def apply_multinomial_nb(class_documents, vocab, prior, condprob,
    test_documents):
  def extract_tokens_from_doc(vocab, doc_content):
    return [term for term in doc_content if term in vocab]

  predictions = []
  for doc_id, doc_content in test_documents:
    W = extract_tokens_from_doc(vocab, doc_content)
    scores = defaultdict(float)

    for class_id in class_documents.keys():
      score = math.log(prior[class_id])
      for term in W:
        score += math.log(condprob[class_id][term])
```

```
14
15          scores[class_id] = score
16
17      predicted_class = max(scores, key=scores.get)
18      predictions.append((doc_id, predicted_class))
19
20  return predictions
```

4.4 Save prediction

After classification, the predicted classes for the test documents are written to a CSV file called `predictions.csv`. Each document ID and its predicted class are saved in the file for later evaluation.

5 Discussion and Conclusion

In this study, I experimented with both log-likelihood and chi-square methods for feature selection. I found that the chi-square method provided better performance compared to log-likelihood. To evaluate the impact of different feature selection sizes, I tested several configurations with 500, 400, 300, and 250 features. The results are as follows:

- 500 features: Score = 0.97470
- 400 features: Score = 0.98326
- 300 features: Score = 0.99590
- 250 features: Score = 0.99093

From these results, it is clear that using 300 features yielded the highest score, suggesting that a balance between model complexity and accuracy can be achieved at this level. However, as the number of features is reduced further, performance starts to degrade, indicating that too few features may lead to the loss of important information.

Additionally, I experimented with different preprocessing strategies. When punctuation and digit removal were excluded from preprocessing, the performance for 300 features resulted in a score of 0.97060. This suggests that while preprocessing (such as punctuation and digit removal) does improve performance, it is not always crucial. However, removing these elements appears to provide a modest improvement in the results, indicating the importance of careful text preprocessing for feature selection.

BEST RESULT ON KAGGLE: Score = 0.99590