Feature Generation from Text Collections using TFIDF Representation

Student *name*
ID *123*
Course *course name*

1 Preprocessing

1.1 Read and Split

Reading datasets from files and directories is the basic work of the text categorization task. The datasets used in this experiment are stored in five subdirectories, where the names of the subdirectories correspond to the tag names, and the files in the subdirectories are text files of the corresponding categories. For the sake of possible subsequent categorization tasks, both text content and labels should be loaded at the same time when loading. So, I use **list** to store the text data and labels in this step, and also use a **dict** type to store the mapping from labeled text to values. The values corresponding to the labels are simply noted as their index order.

Firstly, use **os.listdir()** to get all the subfolders and labels:

```python
import os
labels = os.listdir("./dataset")    # get all labels
l2i = {labels[i]: i for i in range(len(labels))}    # label to index
texts, y = [], []
print("labels:")
print(labels)
print("labels to index")
print(l2i)
```

✓ 0.0s Python

```
labels:
['alt.atheism', 'comp.graphics', 'rec.motorcycles', 'soc.religion.christian', 'talk.politics.misc']
labels to index
{'alt.atheism': 0, 'comp.graphics': 1, 'rec.motorcycles': 2, 'soc.religion.christian': 3, 'talk.politics.misc': 4}
```

Figure 1: load labels

As shown in Figure 1, I got all the labels successfully in this step.

Next, for each label, use **os.listdir** separately to get all the files contained in the subdirectory and use **codecs.open()** to read the text content. For garbled raw text, I use **str.split()** to split it into individual words.

As shown in Figure 2, I managed to load all the text files under the labels, and each document was converted into a list containing one single word. In this step, I also loaded the corresponding numeric labels y.

```
import codecs

for l in labels:
    files = os.listdir(f"dataset/{l}")
    for f in files:
        path = f"dataset/{l}/{f}"
        # read txt file
        file = codecs.open(path, 'r', 'Latin1') # use Latin1 encoding
        content = file.read()
        # split into words
        content = content.lower().split()
        texts.append(content)
        y.append(l2i[l])    # numerical label
print("number of documents", len(texts))
print(texts[0])
print(y[0])
```

✓ 0.6s Python

```
number of documents 2726
['from:', 'mathew', '<mathew@mantis.co.uk>', 'subject:', 'alt.atheism', 'faq:', 'atheist', 'resources', 'summary:', 'books,', '
0
```

Figure 2: load texts

1.2 Remove Stopwords

In this step, I use **re.sub()** to remove non alphabetical characters in a regular expression matching pattern. In this way, each document is given a list consisting entirely of words.The next step is to remove stopwords in the sentence that have little or no point of use in conveying the meaning. In this experiment, the stopwords have been given in a txt file. Notice that this step involves a lot of looking up whether a word appears in the list of stopwords or not. Therefore, I converted the stopwords to the type **set**, a hash-table based data structure that enables efficient data lookups.

```
import re
# load stopwords from txt file
with open(r"stopwords.txt", "r", encoding="utf-8") as fp:
    stopwords = fp.read().splitlines()
stopwords = set(stopwords)
print("number of stopwords", len(stopwords))
print("before removing stopwords:", len(texts[0]))
texts_rs = []
for text in texts:
    # remove non-alphabetic characters
    text_rs = []
    for word in text:
        word = re.sub(r"[^a-z]", " ", word).strip()
        if word != "":
            text_rs.extend(word.split())
    text_rs = [w for w in text_rs if w not in stopwords]
    texts_rs.append(text_rs)
print("after removing stopwords:", len(texts_rs[0]))
print(texts_rs[0])
```

✓ 1.2s Python

```
number of stopwords 851
before removing stopwords: 1704
after removing stopwords: 941
['mathew', 'mathew', 'mantis', 'uk', 'subject', 'alt', 'atheism', 'faq', 'atheist', 'resources', 'summary', 'books', 'addresses'
```

Figure 3: remove stopwords

When deleting a non-alphabetical character from a word, I initially delete the character directly. However, I find this often turns the part into a longer meaningless word, such as [Alt.Atheism], which becomes [altatheism]. It would be more sensible to separate the words before and after the non-alphabetical characters, which would result in two words: [alt atheism].

As can be seen from the results, the number of stopwords is 851. After this step, the

number of words contained in the first document in the dataset decreases from 1704 to 941.

1.3 Word Stemming

The goal of word stemming is to simplify the words to their basic forms, including removing the tense changes of verbs, plural forms of nouns, etc. I implemented this with the help of **SnowballStemmer** from the nltk library[1].

```python
from nltk.stem import SnowballStemmer

stemmer = SnowballStemmer("english")    # Snowball stemmer

texts_stem = []
for t in texts_rs:
    texts_stem.append([stemmer.stem(w) for w in t])

t1, t2 = texts_rs[0], texts_stem[0]
print("length of text:", len(t1))
print("change of stemming:", len([i for i in range(len(t1)) if t1[i] != t2[i]]))
```

✓ 5.1s Python

```
length of text: 941
change of stemming: 456
```

Figure 4: word stemming

In Figure 4, for a document of length 941, the stemming step changed 456 of the words.

2 TFIDF Representation

To achieve better code compactness and readability, I encapsulated the TFIDF word vectorization functionality into a single class. This class takes as input the list of document words preprocessed in the previous step and outputs the document-by-word matrix.

The main properties of this class are defined as follows:

- *document_freq*: (dict), represents the number of document occurrences of the word in the dataset.
- *vocab_table*: (dict), denotes the index of the word in the second dimension of the document-by-word matrix.
- *num_docs*: (int), number of documents in the dataset.
- *dims*: (int), total number of words in the dataset

Getting the above information from the dataset before formally calculating tfidf will greatly facilitate subsequent calculations.

Figure 5: class attributes

As shown in Figure 5, the dataset contains 2726 documents, with a total of 80,491 different words. The *vocab_table* contains a mapping of each word to a numeric index.

2.1 Word Frequency

When calculating the frequency, I processed each document individually, counting the occurrences of each word. This is done by initializing a zero-filled array with dimensions [number of documents, number of unique words]. Then, for each word in each document, I find the index in the vocab_table and increment the corresponding position in the array. After the traversal is completed, this array is accumulated in the second dimension to obtain the total number of words in each document, and the total number is used to calculate the frequency.

```
word_freq = np.zeros((self.num_docs, self.dims)
                     )    # word frequency array

for i, doc in enumerate(docs):
    for word in doc:
        word_freq[i, self.vocab_table[word]] += 1   # count
# calculate frequency
tf = word_freq / word_freq.sum(axis=1, keepdims=True)
```

Figure 6: word frequency

2.2 Document Frequency

In this step, I obtained the document frequency of each word in the dataset. This information is represented via a dict with key as word and value as number of times. For each word in each document, if it is not in the dict, add it to the dict and initialize the count to 1; conversely, the count is incremented by 1.

```python
def update_vocab(self, docs):
    for doc in docs:
        self.num_docs += 1
        words = set(doc)    # keep unique words
        for w in words:
            # for a word, count the num of documents that contain it
            self.document_freq[w] = self.document_freq.get(w, 0) + 1
    self.vocab_table = {w: i for i, w in enumerate(
        self.document_freq.keys())}  # word to index table
    self.dims = len(self.document_freq)  # number of unique words

def _idfs(self):
    # convert dict value to array
    doc_freq = np.array(list(self.document_freq.values()))
    # calculate idf
    return np.log(self.num_docs / (doc_freq))
```

Figure 7: document frequency

2.3 Normalize

TF-IDF is calculated using formula 1:

$$a_{ik} = log(f_{ik} + 1.0) * log(\frac{N}{n_k}) \tag{1}$$

I used the array representation in numpy when calculating tf and idf, so there is no need to calculate element by element in this step, just multiply directly using the **broadcast** mechanism in numpy.

In regularization process, I use numpy to get the sum of squares of the elements in each row of the array and broadcast the division to each element.

```
print("shape of tf-idf matrix:", Aik.shape)
print("rate of 0 in matrix",np.sum(Aik == 0)/Aik.size)
✓ 0.4s                                                    Python
shape of tf-idf matrix: (2726, 80491)
rate of 0 in matrix 0.9977799954175234
```

Figure 8: TF-IDF result

The results obtained are shown in Figure 8. Two things are worth noting:

1. the result is quite **high-dimensional**, with a matrix of the shape (2726, 80491)
2. the result is highly **sparse**, with up to 99% of the transformed word vector matrix is 0

The reason for these two occurrences may be caused by using all the words in the dataset, which can lead to unnecessary storage and computational consumption. Therefore, I tried to remove the low-frequency words from the dataset and encode using only

some of the high-frequency words. The way to implement this is to build a new class by inheriting the previous class, rewrite some methods, count the word frequencies in the statistics set, and encode only the 5000 words with the highest frequency.

Figure 9: limited dimensions

The results of word vectorization after limiting the dimensionality are shown in Figure 9, where the second dimension of the word matrix is reduced from 80,491 to 5,000, and the ratio of 0 in the result is reduced to 97%. I also observed that **the size of the npz file** obtained by the original method is as large as 1.5GB, and the npz result after restricting the dimensionality is only about 100MB.

3 Conclusion

Text vectorization is the groundwork for various Natural Language Processing (NLP) tasks, including text categorization. In this experiment, I inplemented the popular TF-IDF(Term Frequency-Inverse Document Frequency) vectorization method and successfully converted the ONE dataset into a word vector matrix. By calculating the word frequencies within documents and document frequencies across the corpus, TF-IDF achieves straightforward and efficient word vectorization. The result is a highly sparse matrix, which effectively captures the relative importance of words in the dataset. During this experiment, I observed that utilizing the entire vocabulary of the dataset for TF-IDF tends to bring about high memory consumption, which can be alleviated by using a subset of high-frequency words.

References

[1] NLTK Project, sample usage for stem. `https://www.nltk.org/howto/stem.html`. Accessed: 2024-08-19.