به نام خدا

دانشگاه تهران

دانشکدگان فنی

دانشکده مهندسی برق و کامپیوتر

# درس پردازش زبان طبیعی

## پاسخ تمرین{2}

نام و نام خانوادگی: مریم حسینعلی

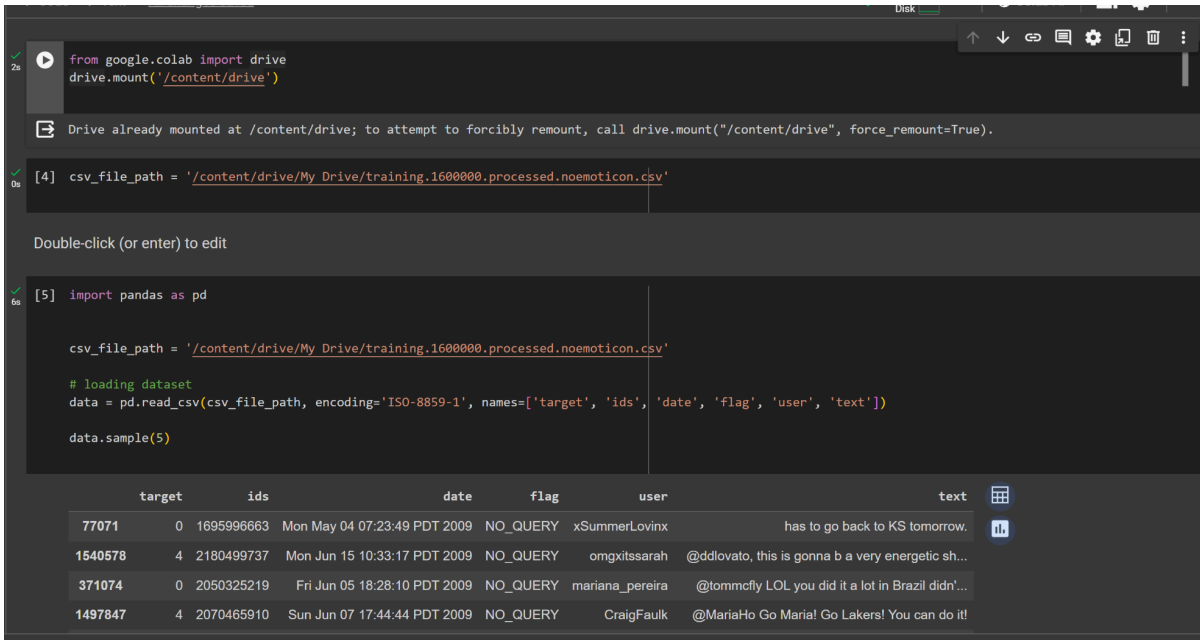
شماره دانشجویی: 610398209

## QUESTION 1

### PART 1 - DATASET PREPROCESSING

First, I uploaded the dataset into my Google Drive to access it in Google Colab.

Then I used pandas to load a dataset from a CSV file that contains tweets. Then, by using ISO-8859-1, I made sure the file was read correctly, and then I provided related names for each column in the dataset. Finally, by printing 5 samples, I checked that the data had been correctly extracted:



```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

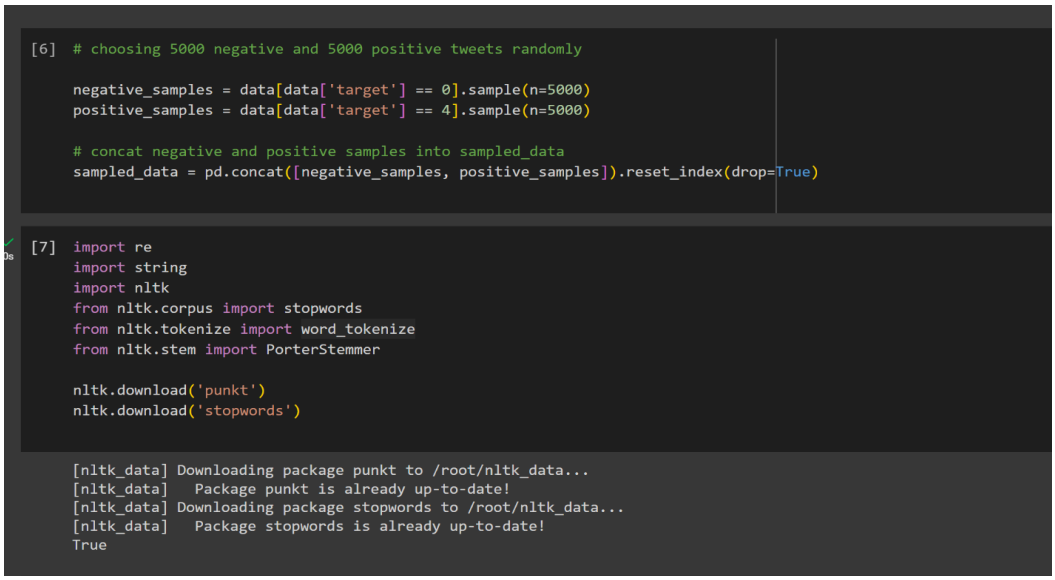
```
[4] csv_file_path = '/content/drive/My Drive/training.1600000.processed.noemoticon.csv'
```

Double-click (or enter) to edit

```
[5] import pandas as pd


    csv_file_path = '/content/drive/My Drive/training.1600000.processed.noemoticon.csv'

    # loading dataset
    data = pd.read_csv(csv_file_path, encoding='ISO-8859-1', names=['target', 'ids', 'date', 'flag', 'user', 'text'])

    data.sample(5)
```

| | target | ids | date | flag | user | text |
|---|---|---|---|---|---|---|
| 77071 | 0 | 1695996663 | Mon May 04 07:23:49 PDT 2009 | NO_QUERY | xSummerLovinx | has to go back to KS tomorrow. |
| 1540578 | 4 | 2180499737 | Mon Jun 15 10:33:17 PDT 2009 | NO_QUERY | omgxitssarah | @ddlovato, this is gonna b a very energetic sh... |
| 371074 | 0 | 2050325219 | Fri Jun 05 18:28:10 PDT 2009 | NO_QUERY | mariana_pereira | @tommcfly LOL you did it a lot in Brazil didn'... |
| 1497847 | 4 | 2070465910 | Sun Jun 07 17:44:44 PDT 2009 | NO_QUERY | CraigFaulk | @MariaHo Go Maria! Go Lakers! You can do it! |

Next, I selected 5000 positive and 5000 negative tweets randomly, using the "target" column to distinguish them (where 0 indicates a negative tweet and 4 indicates a positive one). After that, I imported the necessary libraries for preprocessing:



```
[6]  # choosing 5000 negative and 5000 positive tweets randomly

     negative_samples = data[data['target'] == 0].sample(n=5000)
     positive_samples = data[data['target'] == 4].sample(n=5000)

     # concat negative and positive samples into sampled_data
     sampled_data = pd.concat([negative_samples, positive_samples]).reset_index(drop=True)
```

```
[7]  import re
     import string
     import nltk
     from nltk.corpus import stopwords
     from nltk.tokenize import word_tokenize
     from nltk.stem import PorterStemmer

     nltk.download('punkt')
     nltk.download('stopwords')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
True
```

Next, I defined a preprocessing function to clean and prepare the tweet text for analysis. each part of the function and why its needed:

1. Convert Text to Lowercase:

- Purpose: Converts all text to lowercase to ensure uniformity and prevent the same words in different cases from being treated as distinct.
- Why It's Needed: Text data often contains variations in capitalization, for many analysis tasks, these variations are irrelevant. Lowercasing helps in reducing the complexity of the text data.

2. Remove URLs:

- Purpose: Strips out web addresses, which are common in tweets and other forms of social media text but usually irrelevant to sentiment analysis.
- Why It's Needed: URLs in text data can introduce noise since they do not carry sentiment and are highly variable.

3. Remove Punctuation:

- Purpose: Eliminates punctuation marks from the text, as they are unnecessary for understanding sentiment.
- Why It's Needed: Punctuation removal helps simplify the text data, focusing the analysis on words.

4. Tokenization:

- Purpose: Splits text into individual words or tokens, facilitating further processing like stopword removal and stemming.
- Why It's Needed: Tokenization transforms text from a string into a list of words, enabling word-level analysis and processing. It's a foundational step for most NLP tasks.

5.Remove stopwords:

- Purpose: Filters out common words like "the", "is", … that appear frequently in the language but often don't contribute to the sentiment.
- Why It's Needed: Stopwords can weaken the focus on important words in text analysis. Taking them out helps sentiment analysis models do better by paying more attention to words that matter more..

6. Stemming:

- Purpose: Reduces words to their root form, helping to consolidate variations of a word into a single representation.
- Why It's Needed: Stemming helps in reducing the complexity of the text and increases the chance that different forms of the same word are interpreted as equivalent, aiding in generalization.

7. Rejoin Words:

- Purpose: Combines the processed words back into a single text string.
- Why It's Needed: This step prepares the text for further analysis ensuring that the preprocessing modifications are applied effectively throughout the analysis pipeline.

preprocessing function:

```python
def preprocess_text(text):
  # converting text to lowercase
  text = text.lower()

  # removing URLs
  text = re.sub(r'http\S+|www.\S+', '', text)

  # removing punctuation
  translator = str.maketrans('', '', string.punctuation)

  text = text.translate(translator)

  # tokenization
  words = word_tokenize(text)

  # removing stopwords
  filtered_words = [word for word in words if word not in stopwords.words('english')]

  # stemming
  stemmer = PorterStemmer()
  stemmed_words = [stemmer.stem(word) for word in filtered_words]

  # rejoin words to form the processed text
  return ' '.join(stemmed_words)
```

After that, I applied this function to the `sampled_data` and printed a few samples to ensure the preprocessing was executed correctly:

```python
sampled_data['processed_text'] = sampled_data['text'].apply(preprocess_text)

print(sampled_data[['text', 'processed_text']].head())
```

```
                                                text  \
0  Watching the premier of Kendra, making study n...
1                              Weekend almost done
2  Defeated  ive been reading all my required rea...
3                                has a headache..
4  Is killing it again today! My feet kill though...

                                      processed_text
0  watch premier kendra make studi notecard go be...
1                             weekend almost done
2  defeat ive read requir read cant seem rememb a...
3                                          headach
4  kill today feet kill though post work pint ton...
```

splitting data:

```
4  kill today feet kill though post work pint ton...

[10] from sklearn.model_selection import train_test_split

     # splitting the dataset
     X = sampled_data['text']  # features
     y = sampled_data['target']  # labels

     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

PART2

Transforming Text into Binary Term Frequency Vectors

In this part  I turned the cleaned text into a numerical representation.

I first generated a vocabulary list. This list consists of unique words found across all documents, ensuring that each word has a specific position in the forthcoming vectors. The creation of such a vocabulary is crucial as it serves as a reference for encoding the documents.

Following the vocabulary setup, I initiated the construction of binary term frequency (TF) vectors. These vectors are simple yet powerful; they indicate the presence or absence of words from the vocabulary in each document. For each document, a vector was created where each position corresponds to a word in the vocabulary. If a word is present in the document, its corresponding position in the vector is marked as 1; otherwise, it remains 0. This method results in a binary representation of the document's content, relative to the overall vocabulary.

The binary vectors were then compiled into a pandas DataFrame, named `binary_tf_df`.

```python
import pandas as pd

# converting the processed_text column to a list of documents
documents = sampled_data['processed_text'].tolist()

# creating a sorted list of unique words in the documents to form the vocabulary
vocabulary = sorted(set(word for document in documents for word in document.split()))

# an empty list for binary TF vectors
binary_tf_matrix = []

 # binary TF matrix
for document in documents:

    # a set of unique words in the current document
    words_in_document = set(document.split())

    # setting the binary TF vector with zeros
    binary_tf_vector = [0] * len(vocabulary)

    # iterating over vocab and when a word is present in the document,
    # setting the corresponding value to 1
    for index, word in enumerate(vocabulary):
        if word in words_in_document:
            binary_tf_vector[index] = 1


    binary_tf_matrix.append(binary_tf_vector)

# converting the binary TF matrix to a DataFrame
binary_tf_df = pd.DataFrame(binary_tf_matrix, columns=vocabulary)
```

## PART3

calculating TF-IDF Vectors:

In this part I adopted the TF-IDF technique to refine the text's numerical representation, providing a richer context for each word's significance across the corpus. This method not only considers the term's frequency within a single document (TF) but also how unique the term is across all documents (IDF), thereby offering a balanced measure of word importance.

**Process Overview:**

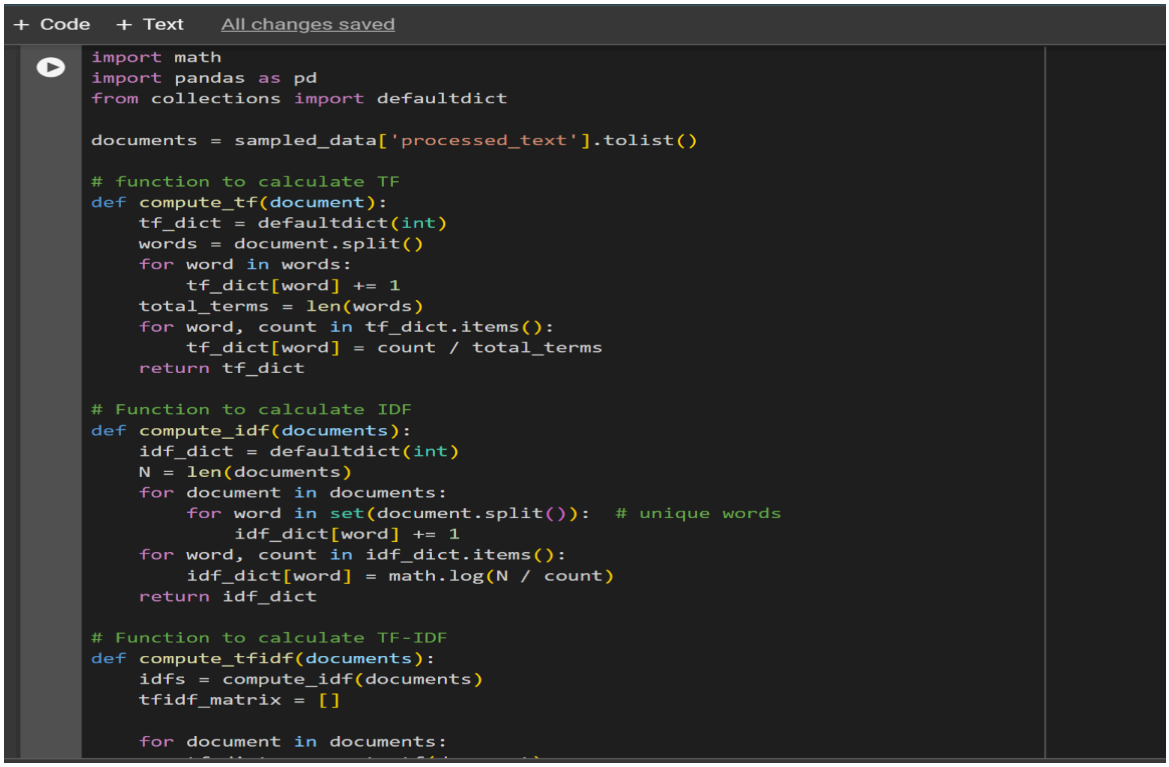
Building the Vocabulary: The process began with identifying all unique words across the documents to create a comprehensive vocabulary. This critical step ensured that each word was given a unique index, laying the groundwork for consistent document encoding.

Calculating TF-IDF Vectors: The computation of TF-IDF vectors involved several key calculations:

- Term Frequency : For every document, I computed the frequency of each word, adjusting these counts by the document's total word count to obtain the term frequency. This adjustment accounts for the varying lengths of documents.
- Inverse Document Frequency: I then calculated each word's IDF value across the entire set of documents. The IDF formula, log(N / n), where N is the total number of documents and n is the number of documents containing the term, highlights words that are rare across the corpus by assigning them higher scores.
- TF-IDF Score: Multiplying a term's TF by its IDF score yields its TF-IDF value. This score represents the term's significance within a particular document, relative to its commonality across all documents. High TF-IDF scores indicate terms that are both prevalent in a document and rare across the dataset, underscoring their potential relevance to the document's meaning.

With TF-IDF vectors calculated for each document, these were then structured into a dense matrix aligned with the vocabulary indices. Each document's vector in this matrix contains the TF-IDF scores for the corresponding words, creating a uniform representation of the text data.

The dense TF-IDF matrix was converted into a pandas DataFrame, tfidf_df, making the data accessible for analysis and modeling. This DataFrame organizes the data in a clear format, allowing for efficient processing in subsequent stages.



```python
# Function to calculate TF-IDF
def compute_tfidf(documents):
    idfs = compute_idf(documents)
    tfidf_matrix = []

    for document in documents:
        tf_dict = compute_tf(document)
        tfidf_vector = {word: tf * idfs[word] for word, tf in tf_dict.items()}
        tfidf_matrix.append(tfidf_vector)

    return tfidf_matrix
# computing the TF-IDF matrix
tfidf_matrix = compute_tfidf(documents)

# convert matrix into a dense matrix
# creating a sorted list of unique words in the documents to form the vocabulary
vocabulary = sorted(set(word for document in documents for word in document.split()))

# create a mapping of vocabulary words to their indexes
vocab_index = {word: i for i, word in enumerate(vocabulary)}

# a matrix to store the TF-IDF vectors
dense_tfidf_matrix = []
for doc_vector in tfidf_matrix:

    doc_tfidf_vector = [0] * len(vocabulary)
    for word, tfidf_value in doc_vector.items():
        index = vocab_index[word]
        doc_tfidf_vector[index] = tfidf_value
    dense_tfidf_matrix.append(doc_tfidf_vector)

# Convert the dense TF-IDF matrix to a DataFrame
tfidf_df = pd.DataFrame(dense_tfidf_matrix, columns=vocabulary)
```

## PART4

creating PPMI Vectors

In this part, I worked on making the text data more meaningful for analysis by using Positive Pointwise Mutual Information (PPMI). PPMI is a way to measure how often two words appear together compared to how often they appear separately. The idea is that words appearing together more than expected by chance are likely to have a strong association.

**Process Overview:**

Counting Words and Pairs: I started by counting how often each word appears in the whole dataset and how often pairs of words appear close to each other within a certain range in the texts.

Calculating Probabilities: With those counts, I looked at the probability of finding word pairs together versus finding them separately across all documents. This helps to see if some words have a special connection in the context of the dataset.

Using the PPMI Formula: To calculate the PPMI values, I applied a formula that contrasts the actual probability of encountering a pair of words together against the probability of their random co-occurrence. The formula is:

$$\mathrm{PMI}(w_1, w_2) = \log_2 \frac{P(w_1, w_2)}{P(w_1)P(w_2)}$$

$$P(w) = \frac{Freq(w)}{totalWordCount}$$

PPMI = PMI if PMI > 0

PPMI = 0 otherwise

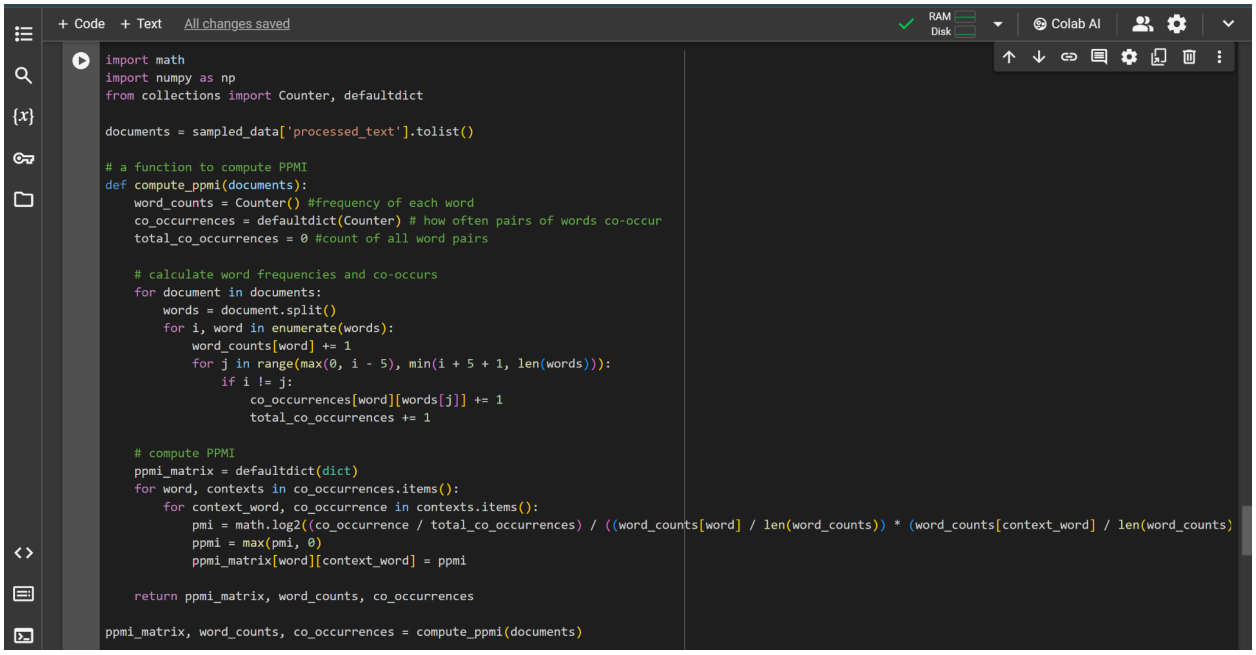
Where:

-(P(w1, w2) represents the probability of both words appearing together.

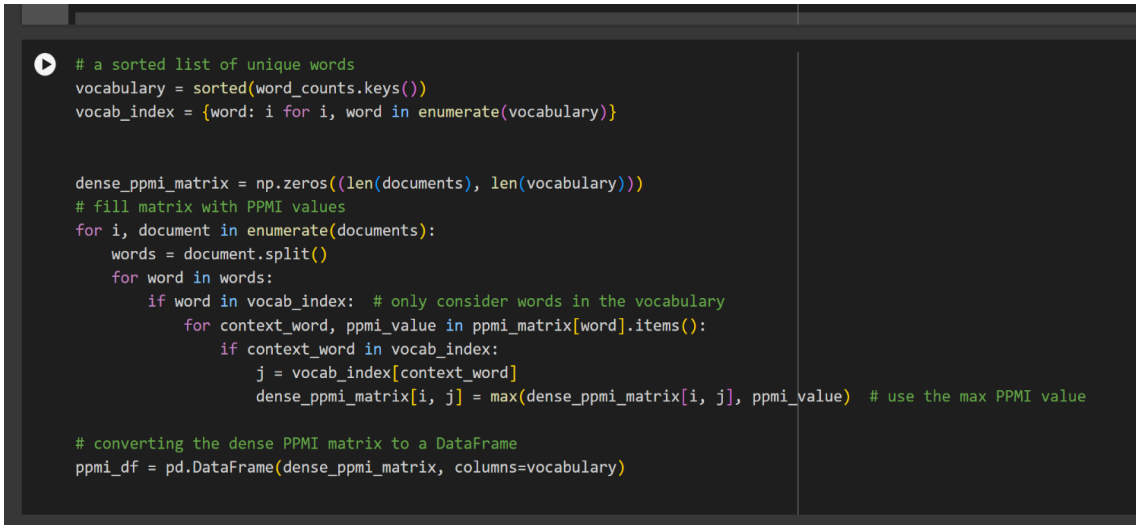
- (P(w1) and (P(w2) are the probabilities of each word appearing in the documents.

If a word pair is found together more frequently than by random chance, it receives a high PPMI score. Conversely, if they occur together as frequently as or less than expected by chance, the PPMI score is low or zero. This method effectively highlights words that share a significant connection within the text.

Creating a Matrix: After calculating the PPMI scores for word pairs, I organized these scores into a matrix, where each row represents a document and each column a word from the vocabulary. This matrix fills up with the PPMI scores, showing the significant connections between words in the context of the documents.




```python
import math
import numpy as np
from collections import Counter, defaultdict

documents = sampled_data['processed_text'].tolist()

# a function to compute PPMI
def compute_ppmi(documents):
    word_counts = Counter() #frequency of each word
    co_occurrences = defaultdict(Counter) # how often pairs of words co-occur
    total_co_occurrences = 0 #count of all word pairs

    # calculate word frequencies and co-occurs
    for document in documents:
        words = document.split()
        for i, word in enumerate(words):
            word_counts[word] += 1
            for j in range(max(0, i - 5), min(i + 5 + 1, len(words))):
                if i != j:
                    co_occurrences[word][words[j]] += 1
                    total_co_occurrences += 1

    # compute PPMI
    ppmi_matrix = defaultdict(dict)
    for word, contexts in co_occurrences.items():
        for context_word, co_occurrence in contexts.items():
            pmi = math.log2((co_occurrence / total_co_occurrences) / ((word_counts[word] / len(word_counts)) * (word_counts[context_word] / len(word_counts)
            ppmi = max(pmi, 0)
            ppmi_matrix[word][context_word] = ppmi

    return ppmi_matrix, word_counts, co_occurrences

ppmi_matrix, word_counts, co_occurrences = compute_ppmi(documents)
```



```python
# a sorted list of unique words
vocabulary = sorted(word_counts.keys())
vocab_index = {word: i for i, word in enumerate(vocabulary)}


dense_ppmi_matrix = np.zeros((len(documents), len(vocabulary)))
# fill matrix with PPMI values
for i, document in enumerate(documents):
    words = document.split()
    for word in words:
        if word in vocab_index:  # only consider words in the vocabulary
            for context_word, ppmi_value in ppmi_matrix[word].items():
                if context_word in vocab_index:
                    j = vocab_index[context_word]
                    dense_ppmi_matrix[i, j] = max(dense_ppmi_matrix[i, j], ppmi_value)  # use the max PPMI value

# converting the dense PPMI matrix to a DataFrame
ppmi_df = pd.DataFrame(dense_ppmi_matrix, columns=vocabulary)
```

PART5

## Model Training and Evaluation

In this part, I trained a Naïve Bayes classifier model using TF, TF-IDF, and PPMI vectors. and then I evaluated which embedding technique offers the best performance for sentiment analysis based on F1-score, Precision, and Recall metrics.

**Training with Term Frequency Embedding**

I began by converting the preprocessed text into binary term frequency vectors, where each vector indicates the presence or absence of words from the vocabulary in each document. After aligning these vectors with the sentiment labels and splitting the dataset, I trained the Naïve Bayes model. The evaluation results were as follows:

**-F1-score: 0.7059**

**- Precision: 0.7422**

**- Recall: 0.6731**

These results suggest that the model is reasonably accurate, with a balance between precision and recall, indicating its effectiveness in identifying positive sentiments.

```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import f1_score, precision_score, recall_score
import pandas as pd
import numpy as np


# re-defining vocabulary
vocabulary = sorted(set(word for document in sampled_data['processed_text'].tolist() for word in document.split()))

# if 'binary_tf_df' has not been created , recreate it here
binary_tf_matrix = []

for document in sampled_data['processed_text'].tolist():
    words_in_document = set(document.split())
    binary_tf_vector = [1 if word in words_in_document else 0 for word in vocabulary]
    binary_tf_matrix.append(binary_tf_vector)

binary_tf_df = pd.DataFrame(binary_tf_matrix, columns=vocabulary)

# align 'binary_tf_df' with labels 'y'
X = binary_tf_df.values
y = sampled_data['target']  # Ensure label column

# splitting the dataset into training and testing sets
X_train_tf, X_test_tf, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# initialize and train the Naive Bayes classifier
clf_tf = MultinomialNB()
clf_tf.fit(X_train_tf, y_train)

# predict on the test set and evaluate
y_pred_tf = clf_tf.predict(X_test_tf)
```

```
clf_tf.fit(X_train_tf, y_train)

# predict on the test set and evaluate
y_pred_tf = clf_tf.predict(X_test_tf)

# as'4' is the label for the positive class
print("Performance with TF Embedding:")
print(f"F1-score: {f1_score(y_test, y_pred_tf, pos_label=4)}")
print(f"Precision: {precision_score(y_test, y_pred_tf, pos_label=4)}")
print(f"Recall: {recall_score(y_test, y_pred_tf, pos_label=4)}")
```

```
Performance with TF Embedding:
F1-score: 0.7059447983014863
Precision: 0.7421875
Recall: 0.6730769230769231
```

**Training with TF-IDF Embedding**

Next, I employed TF-IDF vectors, which consider not only the frequency of words within documents but also their importance across the entire dataset. After training the Naïve Bayes classifier with these vectors, the model's performance was evaluated:

**- F1-score: 0.6860**

**- Precision: 0.7303**

**- Recall: 0.6468**

The TF-IDF results show a slight decrease in performance compared to the binary TF model, suggesting that incorporating word importance across documents does not significantly enhance the model's ability to predict sentiment in this context.

```
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import f1_score, precision_score, recall_score



# align 'tfidf_df' with labels 'y'
X = tfidf_df.values
y = sampled_data['target']  # Ensure this is your label column

# pslitting the dataset into training and testing sets
X_train_tfidf, X_test_tfidf, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# initialize and train the Naive Bayes classifier with TF-IDF vectors
clf_tfidf = MultinomialNB()
clf_tfidf.fit(X_train_tfidf, y_train)

# Predict on the test set and evaluate
y_pred_tfidf = clf_tfidf.predict(X_test_tfidf)

# as the label positive class is '4'
print("Performance with TF-IDF Embedding:")
print(f"F1-score: {f1_score(y_test, y_pred_tfidf, pos_label=4)}")
print(f"Precision: {precision_score(y_test, y_pred_tfidf, pos_label=4)}")
print(f"Recall: {recall_score(y_test, y_pred_tfidf, pos_label=4)}")
```

```
Performance with TF-IDF Embedding:
F1-score: 0.6859903381642511
Precision: 0.7302857142857143
Recall: 0.6467611336032388
```

**Training with PPMI Embedding**

Finally, I explored the use of PPMI vectors, aiming to capture the strength of word associations beyond their mere occurrence. The PPMI-based Naïve Bayes model demonstrated the following performance metrics:

**- F1-score: 0.6877**

**- Precision: 0.7041**

**- Recall: 0.6721**

The PPMI model shows an improvement in recall compared to the TF-IDF model, indicating its potential to better capture relevant features for sentiment analysis by emphasizing meaningful word associations.

```
X_ppmi = ppmi_df.values
y = sampled_data['target'].values  # ensure label column

# splitting the dataset into training and testing sets
X_train_ppmi, X_test_ppmi, y_train, y_test = train_test_split(X_ppmi, y, test_size=0.2, random_state=42)

# initialize and train the Naive Bayes classifier with PPMI vectors
clf_ppmi = MultinomialNB()
clf_ppmi.fit(X_train_ppmi, y_train)

# predict on the test set and evaluate
y_pred_ppmi = clf_ppmi.predict(X_test_ppmi)
# # as the label positive class is '4'
print("Performance with PPMI Embedding:")
print(f"F1-score: {f1_score(y_test, y_pred_ppmi, pos_label=4)}")
print(f"Precision: {precision_score(y_test, y_pred_ppmi, pos_label=4)}")
print(f"Recall: {recall_score(y_test, y_pred_ppmi, pos_label=4)}")
```

```
Performance with PPMI Embedding:
F1-score: 0.6877265665458312
Precision: 0.704135737009544
Recall: 0.6720647773279352
```

**analyzing results:**

Term Frequency  Results:

The TF model did a bit better than the others when we checked its F1-score, which tells us how well the model balances being accurate (precision) and catching all the positive sentiments (recall). This suggests that counting words straightforwardly works quite well for understanding sentiments in texts.

TF-IDF Results:

Using TF-IDF, which pays more attention to words that are special or unique in some texts but not all over, didn't improve things much. It seems that for figuring out if a text is positive or

negative, knowing the unique words isn't as helpful as we thought. Maybe the key to sentiment isn't always in the rare words but in the common ones that clearly show emotion.

PPMI Results:

The PPMI approach, which looks at how words pair up in meaningful ways, also didn't beat the simple TF method. This tells us that even though understanding the connection between words is interesting, it might not always help more with figuring out sentiment.

Across the three embedding techniques, the Term Frequency model slightly outperformed the others in terms of F1-score, a measure that balances precision and recall. However, the differences were not substantial, indicating that each method has its merits depending on the specific characteristics of the text data and the goals of the analysis.

colab link:

NLP-CA{2}-610398209.ipynb