电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

实验报告

EXPERIMENT REPORT

STUDENT NAME:	JAHID SHAHIDUL ISLAM
STUDENT ID:	202324090107
COURSE NAME:	PYTHON PRACTICAL PROGRAMMING
TEACHER NAME:	PROF. RAO YUNBO
EXPERIMENT NO:	SIX
DATE:	6th June 2024

1. Experiment title：Install Python Platform

2. Experiment hours：4h Experiment location: Software Building 400

3. Objectives

   At the end of this experiment, you will be able to:

   - At the end of this experiment, you will be able to:
   - How to use Jupyter for SAM.

4. Experimental contents & step

   1) using Jupyter for Natural Language Processing Tasks.
   2) understand twitter_Logistic.ipynb code.
   3) understand bi_lstm.ipynb code.
   4) understand Word2Vec code.
   5) understand GloVe code.

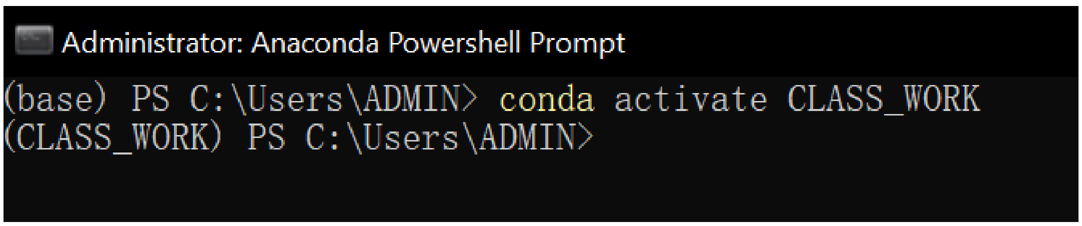
5. Experimental analysis

1. Using Jupyter For Natural Language Processing Tasks.

Step 1: Activate the Conda Environment

I started by activating my Conda environment named "CLASS_WORK," where I wanted to perform the natural language processing (NLP) tasks:

conda activate CLASS_WORK



Step 2: Install SpaCy and Language Models

Within the activated environment, I installed the SpaCy library and language models for Chinese, English, and Spanish:

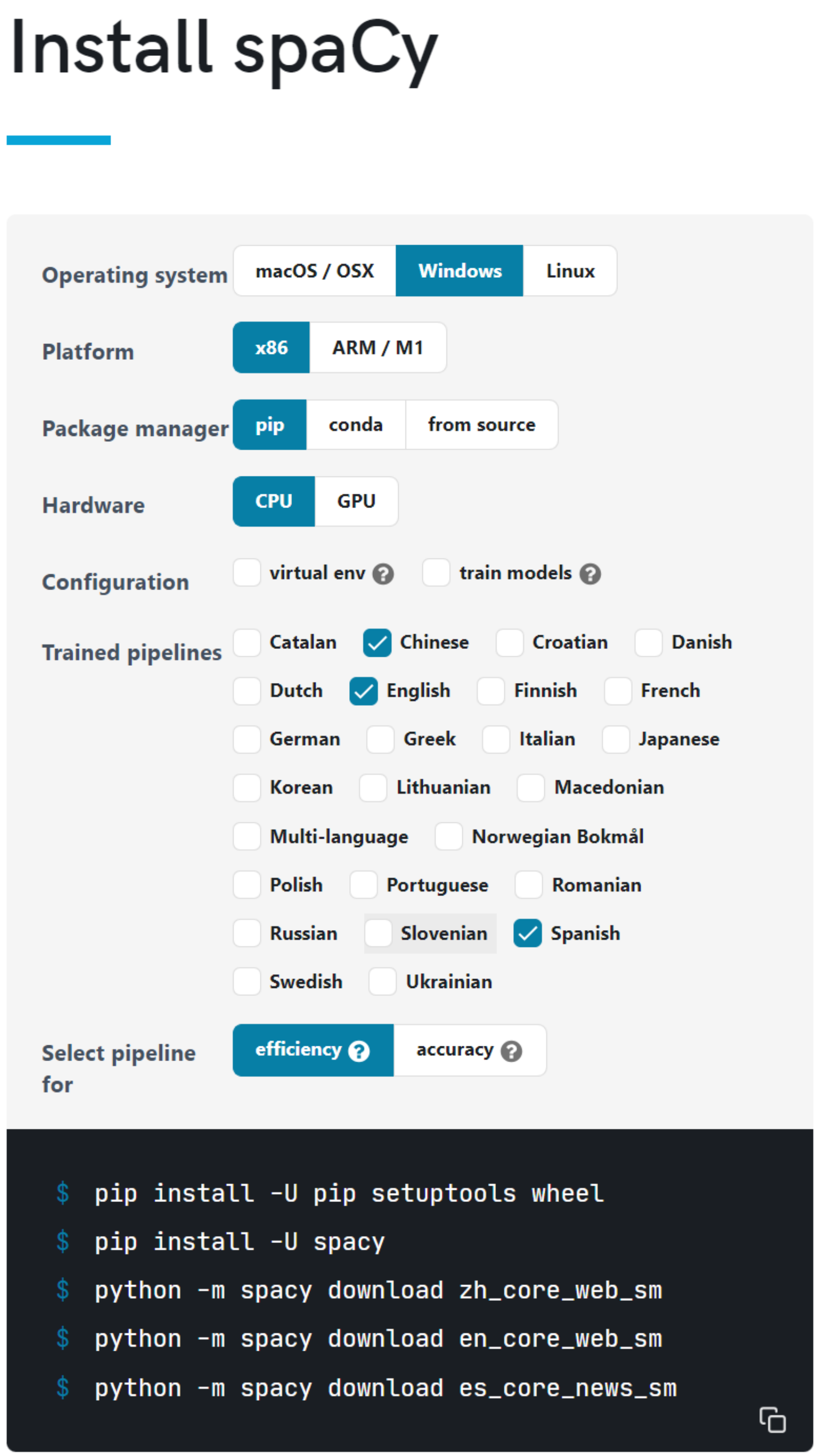
pip install -U pip setuptools wheel

pip install -U spacy

python -m spacy download zh_core_web_sm

python -m spacy download en_core_web_sm

python -m spacy download es_core_news_sm



These language models provide pre-trained word vectors and linguistic annotations for the respective languages.

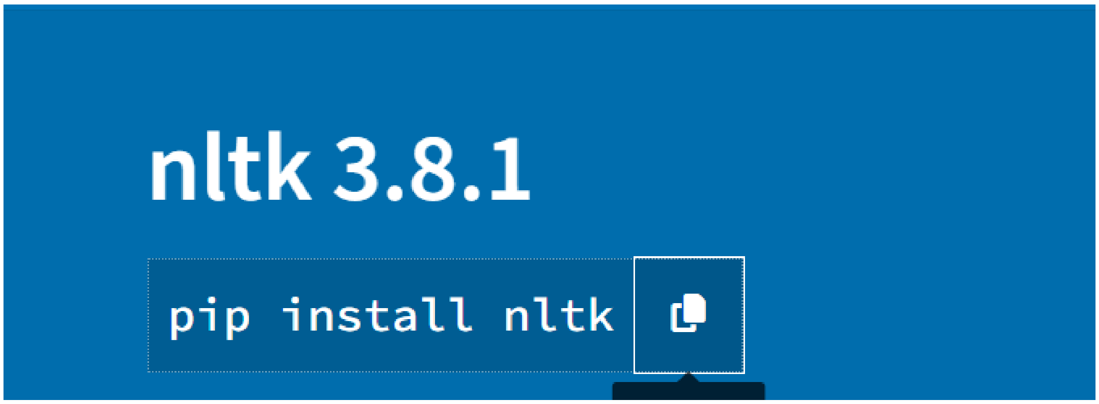
Step 3: Experiment with SpaCy

I imported SpaCy into a Jupyter Notebook and experimented with various NLP tasks, focusing on part-of-speech tagging. I utilized the downloaded language models to analyze text in different languages.

Step 4: Install NLTK and Data

Next, I installed the NLTK library and its data:

pip install nltk



I then downloaded the necessary NLTK data within a Python script or directly in the Jupyter Notebook:

import nltk

nltk.download()

Installing NLTK Data

After installing the NLTK package, please do install the necessary datasets/models for specific functions to work.

If you're unsure of which datasets/models you'll need, you can install the "popular" subset of NLTK data, on the command line type `python -m nltk.downloader popular`, or in the Python interpreter `import nltk; nltk.download('popular')`

For details, see **https://www.nltk.org/data.html**

Step 5: Experiment with NLTK

I used NLTK for part-of-speech tagging, comparing its performance and results with SpaCy.

```python
import nltk
nltk.download('averaged_perceptron_tagger')
```

[1] Python

```
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     C:\Users\ADMIN\AppData\Roaming\nltk_data...
[nltk_data]   Unzipping taggers\averaged_perceptron_tagger.zip.
```

```
True
```

```python
sent= "Here we are learning how does POS Taggingv works"
```

[2] Python

```python
sent= sent.lower()
```

[3] Python

```python
words=nltk.word_tokenize(sent)
```

[4] Python

+ Code + Markdown

```python
words
```

[5] Python

```
['here', 'we', 'are', 'learning', 'how', 'does', 'pos', 'taggingv', 'works']
```

```python
pos_tags=nltk.pos_tag(words)
```

[6] Python

```python
pos_tags
```

[7] Python

```
[('here', 'RB'),
 ('we', 'PRP'),
 ('are', 'VBP'),
 ('learning', 'VBG'),
 ('how', 'WRB'),
 ('does', 'VBZ'),
 ('pos', 'VB'),
 ('taggingv', 'VB'),
 ('works', 'NNS')]
```

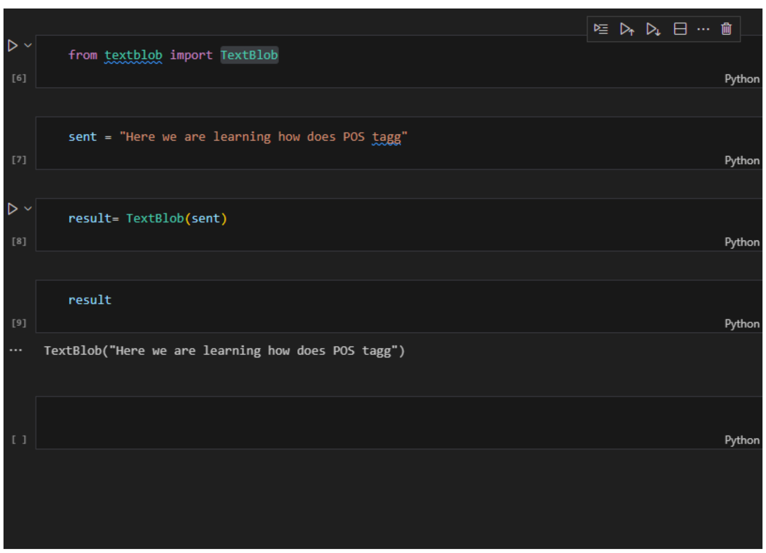
Step 6: Install TextBlob

I installed TextBlob, another popular NLP library:

pip install textblob

Step 7: Experiment with TextBlob

I conducted experiments using TextBlob, including part-of-speech tagging, sentiment analysis, and other NLP tasks supported by the library.



Outcome:

By leveraging Jupyter Notebook, I was able to seamlessly install and utilize various NLP libraries and language models. I conducted experiments with SpaCy, NLTK, and TextBlob, exploring their functionalities and comparing their performance on tasks like part-of-speech tagging. This interactive environment facilitated a comprehensive exploration of NLP concepts and techniques.

2. Understand Twitter_Logistic.Ipynb Code.

This code performs sentiment analysis on Twitter data using a Logistic Regression model. Let's break down the code into 10 steps, focusing on the core parts and mentioning the key functions and modules:

Step 1: Import Libraries

The code begins by importing essential libraries for data processing, visualization, text analysis, and machine learning.

```python
import numpy as np # linear algebra
import pandas as pd # data processing
pd.options.mode.chained_assignment = None
import os #File location
for dirname, _, filenames in os.walk('./'):
    for filename in filenames:
        print(os.path.join(dirname, filename))

from wordcloud import WordCloud #Word visualization
import matplotlib.pyplot as plt #Plotting properties
import seaborn as sns #Plotting properties
from sklearn.feature_extraction.text import CountVectorizer #Data transformation
from sklearn.model_selection import train_test_split #Data testing
from sklearn.linear_model import LogisticRegression #Prediction Model
from sklearn.metrics import accuracy_score #Comparison between real and predicted
from xgboost import XGBClassifier
from sklearn.preprocessing import LabelEncoder #Variable encoding and decoding for XGBoost
import re #Regular expressions
import nltk
from nltk import word_tokenize
# https://github.com/nltk/nltk_data
nltk.download('stopwords')
nltk.download('punkt')
```

Step 2: Load Datasets

The validation and training datasets are loaded using the pd.read_csv() function from the pandas library. These datasets are assumed to be in CSV format.

```
#Validation dataset
val=pd.read_csv("./twitter_validation.csv", header=None)
#Full dataset for Train-Test
train=pd.read_csv("./twitter_training.csv", header=None)
```
[3]

Step 3: Rename Columns and Data Overview

The columns of the datasets are renamed for clarity, and the head() function is used to display the first few rows of each dataset, providing an initial look at the data structure.

对数据中列的名字进行重命名

(variable) train: DataFrame

```
train.columns=['id','information','type','text']
train.head()
```
[4]

	id	information	type	text
0	2401	Borderlands	Positive	im getting on borderlands and i will murder yo...
1	2401	Borderlands	Positive	I am coming to the borders and I will kill you...
2	2401	Borderlands	Positive	im getting on borderlands and i will kill you ...
3	2401	Borderlands	Positive	im coming on borderlands and i will murder you...
4	2401	Borderlands	Positive	im getting on borderlands 2 and i will murder ...

```
val.columns=['id','information','type','text']
val.head()
```
[5]

	id	information	type	text
0	3364	Facebook	Irrelevant	I mentioned on Facebook that I was struggling ...
1	352	Amazon	Neutral	BBC News - Amazon boss Jeff Bezos rejects clai...
2	8312	Microsoft	Negative	@Microsoft Why do I pay for WORD when it funct...
3	4371	CS-GO	Negative	CSGO matchmaking is so full of closet hacking,...
4	4433	Google	Neutral	Now the President is slapping Americans in the...

```
train_data=train
train_data
```
[6]

	id	information	type	text
0	2401	Borderlands	Positive	im getting on borderlands and i will murder yo...
1	2401	Borderlands	Positive	I am coming to the borders and I will kill you...
2	2401	Borderlands	Positive	im getting on borderlands and i will kill you ...

Step 4: Text Preprocessing

The tweet text is preprocessed to prepare it for analysis. This includes converting text to lowercase using str.lower(), ensuring all entries are strings, and removing special characters using regular expressions (re.sub()).

```python
#Text transformation
train_data["lower"]=train_data.text.str.lower() #lowercase
train_data["lower"]=[str(data) for data in train_data.lower] #converting all to string
train_data["lower"]=train_data.lower.apply(lambda x: re.sub('[^A-Za-z0-9 ]+', ' ', x)) #regex
val_data["lower"]=val_data.text.str.lower() #lowercase
val_data["lower"]=[str(data) for data in val_data.lower] #converting all to string
val_data["lower"]=val_data.lower.apply(lambda x: re.sub('[^A-Za-z0-9 ]+', ' ', x)) #regex
```

[8] Pytho

下表显示了这两个文本列之间的差异。

```python
train_data.head()
```

[9] Pytho

	id	information	type	text	lower
0	2401	Borderlands	Positive	im getting on borderlands and i will murder yo...	im getting on borderlands and i will murder yo...
1	2401	Borderlands	Positive	I am coming to the borders and I will kill you...	i am coming to the borders and i will kill you...
2	2401	Borderlands	Positive	im getting on borderlands and i will kill you ...	im getting on borderlands and i will kill you ...
3	2401	Borderlands	Positive	im coming on borderlands and i will murder you...	im coming on borderlands and i will murder you...
4	2401	Borderlands	Positive	im getting on borderlands 2 and i will murder ...	im getting on borderlands 2 and i will murder ...

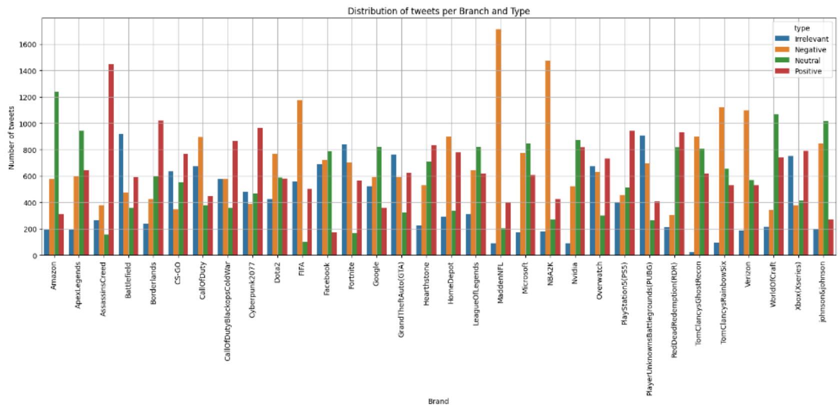
Step 5: Feature Visualization

Word clouds are generated using the WordCloud module to visualize the most frequent words associated with each sentiment category ("Positive", "Negative", "Irrelevant", "Neutral").

```python
word_cloud_text = ''.join(train_data[train_data["type"]=="Positive"].lower)
#Creation of wordcloud
wordcloud = WordCloud(
    max_font_size=100,
    max_words=100,
    background_color="black",
    scale=10,
    width=800,
    height=800
).generate(word_cloud_text)
#Figure properties
plt.figure(figsize=(10,10))
plt.imshow(wordcloud, interpolation="bilinear")
plt.axis("off")
plt.show()
```

Step 6**: Data Distribution Visualization**

A bar plot is created using seaborn.barplot() to show the distribution of tweets across different brands and sentiment categories.



Step 7: Tokenization and Stop Word Removal

The preprocessed text is tokenized using word_tokenize() from the nltk library. The number of unique tokens is calculated to assess the vocabulary size. English stop words are loaded from nltk.corpus.stopwords and stored for later use in removing common words that don't carry much semantic meaning.

使用干净的文本，对分词（token）进行计数，以确定模型的复杂性。目前，有超过3万个不同的单词。

```python
# 将每个sentence分为token的list，所有sentence的token list组成了一个更大的list。
tokens_text = [word_tokenize(str(word)) for word in train_data.lower]
# Unique word counter
tokens_counter = [item for sublist in tokens_text for item in sublist]
print("Number of tokens: ", len(set(tokens_counter)))
```

[18] Python

```
Number of tokens:  30436
```

Step 8: Building the Bag-of-Words Model

A Bag-of-Words (BoW) model is created using CountVectorizer from sklearn.feature_extraction.text. This model converts text into numerical feature vectors based on word frequencies. The tokenizer, stop_words, and ngram_range parameters are used to configure the BoW model.

```python
#Initial Bag of Words
bow_counts = CountVectorizer(
    tokenizer=word_tokenize,
    stop_words=stop_words, #English Stopwords
    ngram_range=(1, 1) #analysis of one word
)
```

Step 9: Training and Evaluating the Logistic Regression Model

The data is split into training and testing sets using train_test_split(). The BoW model is fit to the training data using fit_transform(), and the test data is transformed using the same vocabulary. A Logistic Regression model (LogisticRegression) is trained on the BoW features and evaluated on the test set using accuracy_score().

```
# Logistic regression
model1 = LogisticRegression(C=1, solver="liblinear",max_iter=200)
model1.fit(X_train_bow, y_train_bow)
# Prediction
test_pred = model1.predict(X_test_bow)
print("Accuracy: ", accuracy_score(y_test_bow, test_pred) * 100)
```

27]

Step 10: Validating the Model

The trained Logistic Regression model is applied to the validation dataset to assess its performance on unseen data. The accuracy on the validation set is calculated, and a confusion matrix and classification report are generated using functions from sklearn.metrics to provide a detailed.

```
from sklearn.metrics import accuracy_score, classification_report, confusion_matrix
print(confusion_matrix(y_val_bow, Val_res))
print("\n")
print(classification_report(y_val_bow, Val_res))
```

```
              precision    recall  f1-score   support

  Irrelevant       0.93      0.82      0.87       172
    Negative       0.90      0.95      0.93       266
     Neutral       0.95      0.92      0.93       285
    Positive       0.90      0.94      0.92       277

    accuracy                           0.92      1000
   macro avg       0.92      0.91      0.91      1000
weighted avg       0.92      0.92      0.92      1000
```

3. Understand Bi_Lstm.Ipynb Code.

tep 1: Import Libraries: Import necessary libraries for data manipulation, visualization, text processing, and machine learning, including Pandas, NumPy, Matplotlib, Seaborn, SpaCy, NLTK, TensorFlow, PyTorch, and others.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import spacy
import warnings
import re
import string
import random


from wordcloud import WordCloud
import matplotlib.pyplot as plt
from nltk.tokenize import RegexpTokenizer , TweetTokenizer
from nltk.stem import WordNetLemmatizer ,PorterStemmer
from nltk.corpus import stopwords
from collections import defaultdict
from collections import Counter
from tensorflow.keras.preprocessing.text import one_hot
from tensorflow.keras.preprocessing.sequence import pad_sequences
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader, Dataset
import tensorflow as tf
from tqdm import tqdm
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
import nltk

nlp = spacy.load("en_core_web_sm")
warnings.filterwarnings('ignore')
```

Step 2: Load and Inspect Dataset: Load the Twitter sentiment analysis dataset using pd.read_csv(), examine its shape, column names, and data types, and display a few random samples.

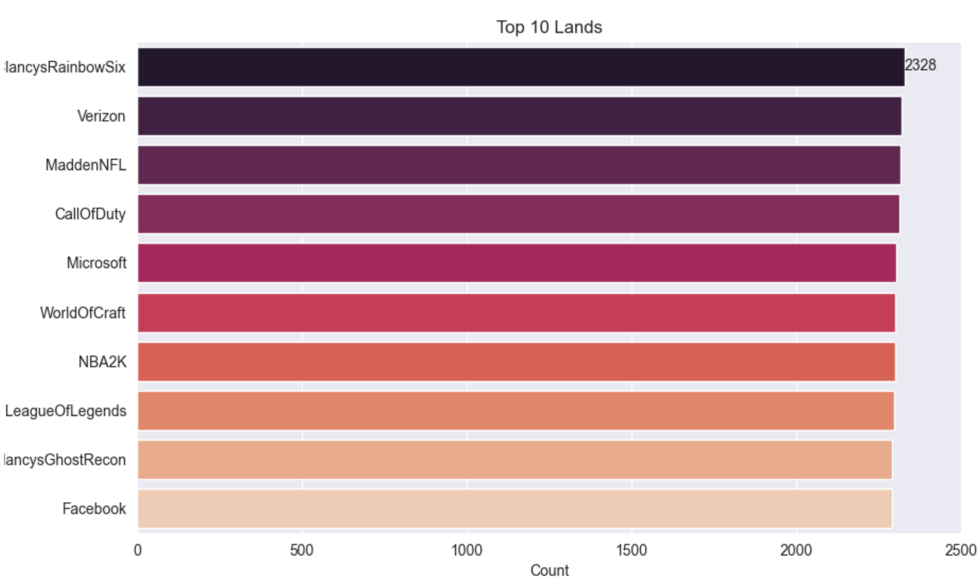
```
df = pd.read_csv('./twitter_training.csv')
✓ 0.0s
```

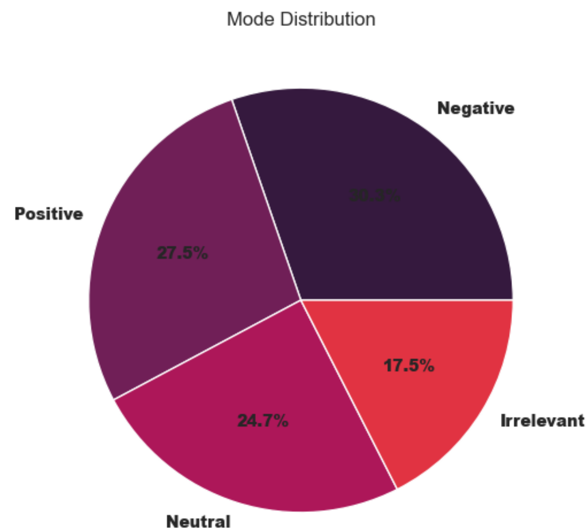
Step 3: Exploratory Data Analysis (EDA): Analyze the dataset for missing values, duplicates, and the distribution of data across different features like "Land" (brand) and "Mode" (sentiment). Visualize these distributions using bar plots and pie charts.

```python
def show_details(dataset):
    missed_values = dataset.isnull().sum()
    missed_values_percent = (dataset.isnull().sum()) / len(dataset)
    duplicated_values = dataset.duplicated().sum()
    duplicated_values_percent = (dataset.duplicated().sum()) / len(dataset)
    info_frame = pd.DataFrame({'Missed_Values' : missed_values ,
                              'Missed_Values %' :missed_values_percent,
                              'Duplicated values' :duplicated_values,
                              'Duplicated values %':duplicated_values_percent})
    return info_frame.T
```
✓ 0.0s

```python
show_details(df)
```
✓ 0.0s Python

	2401	Borderlands	Positive	im getting on borderlands and i will murder you all ,
Missed_Values	0.000000	0.000000	0.000000	686.000000
Missed_Values %	0.000000	0.000000	0.000000	0.009186
Duplicated values	2700.000000	2700.000000	2700.000000	2700.000000
Duplicated values %	0.036154	0.036154	0.036154	0.036154





Step 4: Text Cleaning and Preprocessing: Apply custom functions (clean_emoji() and text_cleaner()) to clean the tweet text, including removing emojis, correcting common contractions, removing URLs and non-alphanumeric characters, and converting text to lowercase.

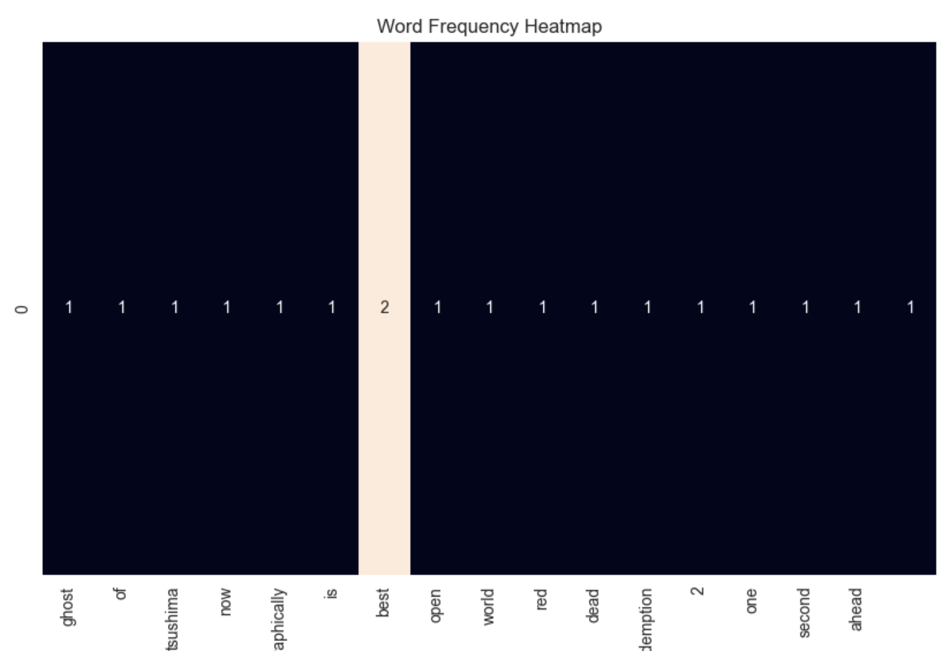
```
def clean_emoji(tx):
    emoji_pattern = re.compile("["
                           u"\U0001F600-\U0001F64F"  # emoticons
                           u"\U0001F300-\U0001F5FF"  # symbols
                           u"\U0001F680-\U0001F6FF"  # transport
                           u"\U0001F1E0-\U0001F1FF"  # flags
                           u"\U00002702-\U000027B0"
                           u"\U000024C2-\U0001F251"
                           "]+", flags=re.UNICODE)

    return emoji_pattern.sub(r'', tx)
def text_cleaner(tx):

    text = re.sub(r"won\'t", "would not", tx)
    text = re.sub(r"im", "i am", text)
    text = re.sub(r"Im", "I am", text)
    text = re.sub(r"can\'t", "can not", text)
    text = re.sub(r"don\'t", "do not", text)
    text = re.sub(r"shouldn\'t", "should not", text)
    text = re.sub(r"needn\'t", "need not", text)
    text = re.sub(r"hasn\'t", "has not", text)
    text = re.sub(r"haven\'t", "have not", text)
    text = re.sub(r"weren\'t", "were not", text)
    text = re.sub(r"mightn\'t", "might not", text)
```

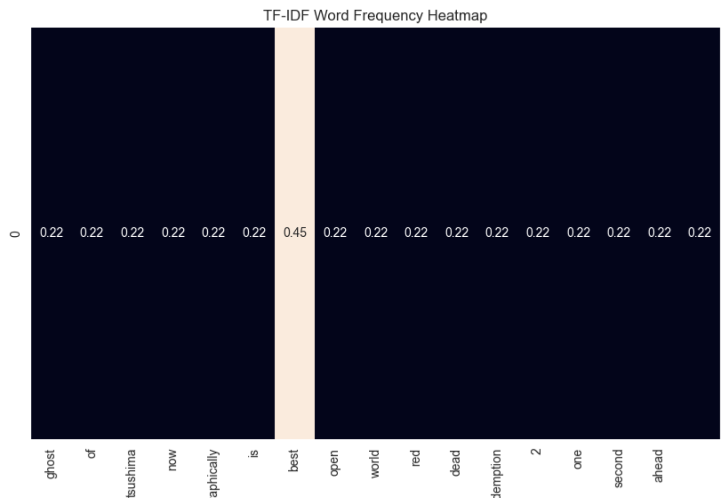
Step 5: Illustrate Common NLP Techniques: Demonstrate various NLP concepts like part-of-speech tagging, named entity recognition, chunking, tokenization, counter vectorization, TF-IDF, and N-grams using examples from the dataset and relevant libraries like SpaCy and NLTK.

```
doc = nlp(test_text)
for token in doc :
    print(f'{token} => {token.pos_}')
```

✓ 0.0s

```
ghost => NOUN
of => ADP
tsushi => PROPN
ama => NOUN
is => AUX
now => ADV
graphically => ADV
the => DET
```



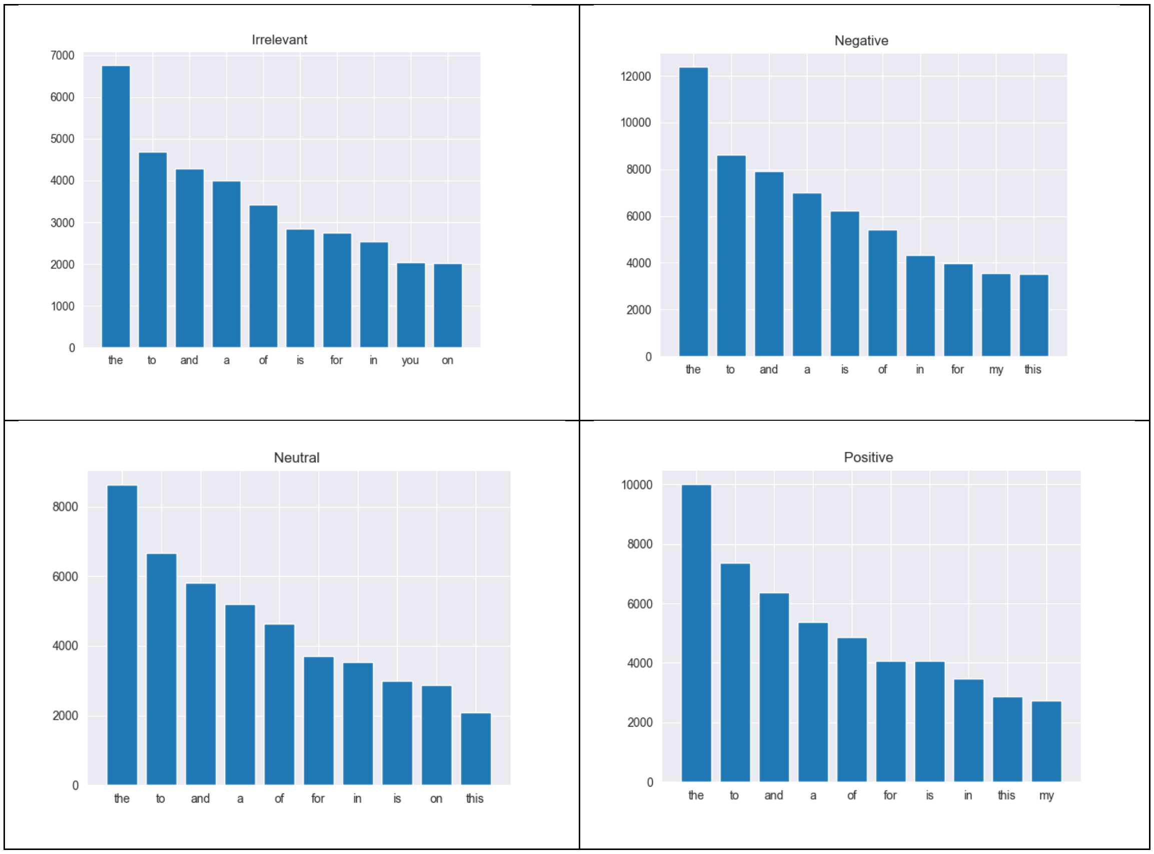


Step 6: Stop Word Analysis and Removal: Analyze the distribution of stop words across different sentiment categories and visualize the most frequent stop words using bar plots. Apply stop word removal to the tweet text using the stopwords_cleaner() function and NLTK's list of stop words.

```
stopwords_list = stopwords.words('english')
print(f'There are {len(stopwords_list) } stop words')
print('**' * 20 , '\n20 of them are as follows:\n')
for inx , value in enumerate(stopwords_list[:20]):
    print(f'{inx+1}:{value}')
```

```
✓ 0.0s
```

```
There are 179 stop words
****************************************
20 of them are as follows:

1:i
2:me
3:my
4:myself
5:we
6:our
7:ours
8:ourselves
9:you
10:you're
11:you've
12:you'll
13:you'd
14:your
15:yours
16:yourself
17:yourselves
18:he
19:him
20:his
```



Step 7: Lemmatization and Stemming: Illustrate the concepts of lemmatization and stemming using SpaCy and the Porter Stemmer from NLTK. Apply stemming to the tweet text to reduce words to their root form.

```
nlp = spacy.load("en_core_web_sm")
doc = nlp(test_text)
for token in doc :
    print(f'{token} => {token.lemma_}')
```

✓ 0.2s

```
ghost => ghost
of => of
tsushi => tsushi
ama => ama
is => be
now => now
graphically => graphically
the => the
best => well
open => open
world => world
. => .
red => red
dead => dead
redemption => redemption
2 => 2
is => be
one => one
second => second
ahead => ahead
. => .
```

```python
# lemmatizer = WordNetLemmatizer()
Stemmer = PorterStemmer()
def stopwords_cleaner(text):
#     word = [lemmatizer.lemmatize(letter) for letter in text if lette
    word = [Stemmer.stem(letter) for letter in text if letter not in s
    peasting = ' '.join(word)
    return peasting
df['Text'] = df['Text'].apply(lambda x : stopwords_cleaner(x))
# stopwords_cleaner(Tokenizer.tokenize(df.Text[100]))
```

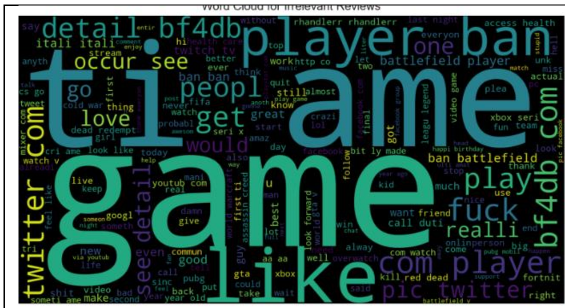
[37] ✓ 8.7s

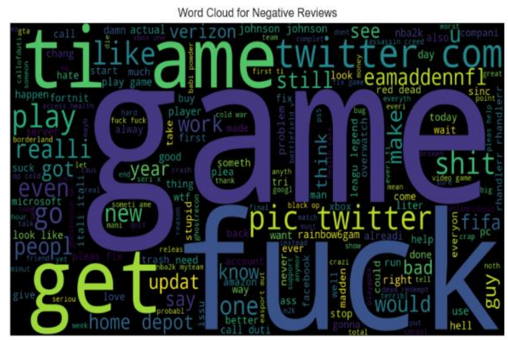
```python
df['Text'][:10].to_frame()
```

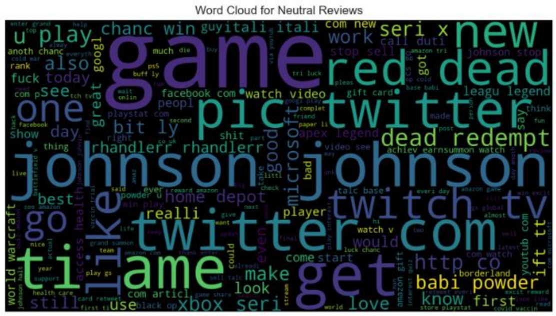
[38] ✓ 0.0s

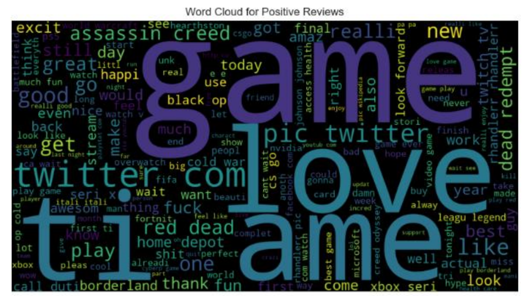
	Text
0	come border kill
1	get borderland kill
2	come borderland murder
3	get borderland 2 murder
4	get borderland murder
5	spent hour make someth fun know huge borderlan...
6	spent coupl hour someth fun know huge borderla...
7	spent hour someth fun know huge borderland fan...
8	spent hour make someth fun know huge rhandlerr...
9	2010 spent hour make someth fun know huge rhan...

Step 8: Word Cloud Visualization: Generate word clouds for each sentiment category using the WordCloud module, providing a visual representation of the most frequent words associated with each sentiment.









Step 9: Prepare Dataset for Deep Learning: Analyze the distribution of tweet lengths and preprocess the text data for input into a Bi-LSTM model. This includes converting text to sequences using Tokenizer.texts_to_sequences(), padding sequences to a fixed length (MAX_LEN), and creating data loaders using PyTorch's DataLoader.

```
tokenizer = tf.keras.preprocessing.text.Tokenizer()
tokenizer.fit_on_texts(df.Text.values.tolist())

xtrain = tokenizer.texts_to_sequences(tra
xtest = tokenizer.texts_to_sequences(test       (function) pad_sequences: Any
xtrain = tf.keras.preprocessing.sequence.pad_sequences(xtrain,maxlen = MAX_LEN)
xtest = tf.keras.preprocessing.sequence.pad_sequences(xtest,maxlen = MAX_LEN)
train_dataset = Dataset(text=xtrain,sentiment=train_df.sentiments.values)
train_loader = torch.utils.data.DataLoader(train_dataset,batch_size=BATCH_SIZE,drop_last=True)
valid_dataset = Dataset(text=xtest,sentiment=test_df.sentiments.values)
valid_loader = torch.utils.data.DataLoader(valid_dataset,batch_size=BATCH_SIZE,drop_last=True)
```

Step 10: Build, Train, and Evaluate the Bi-LSTM Model: Define a Bi-LSTM model architecture (sentimentBiLSTM) using PyTorch. Load pre-trained word embeddings (GloVe), train the model using an Adam optimizer and cross-entropy loss, and evaluate its performance on training and validation sets. Visualize the training and validation accuracy and loss over epochs.

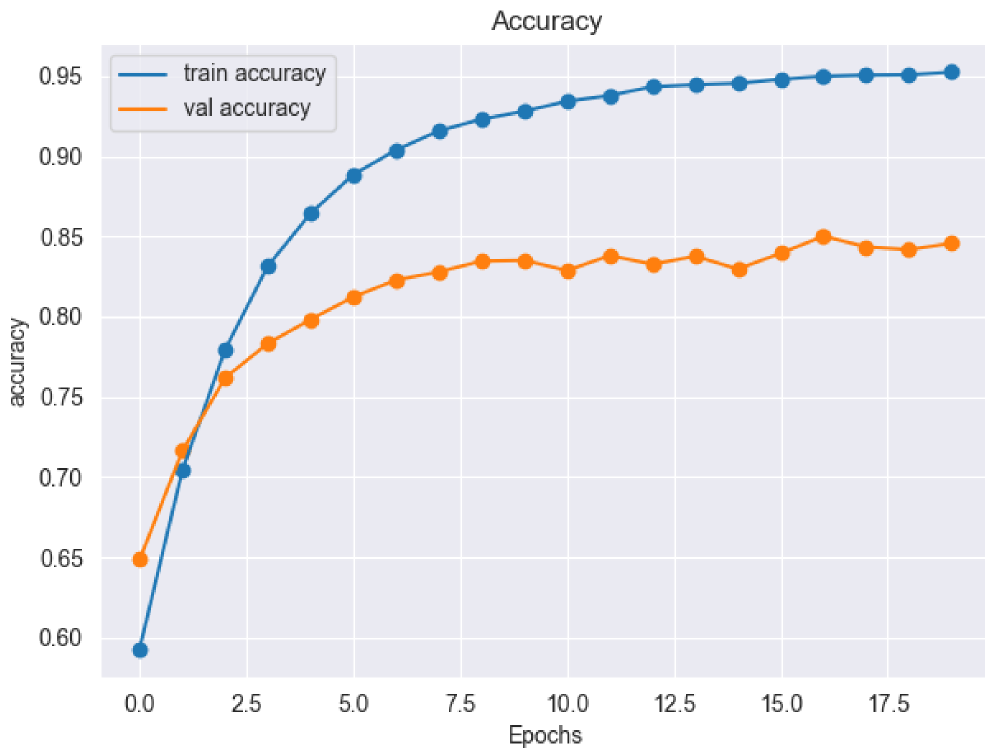
```python
# 实例化模型
model = sentimentBiLSTM(embedding_matrix,hidden_dim,output_size)
model = model.to(device)
print(model)
```

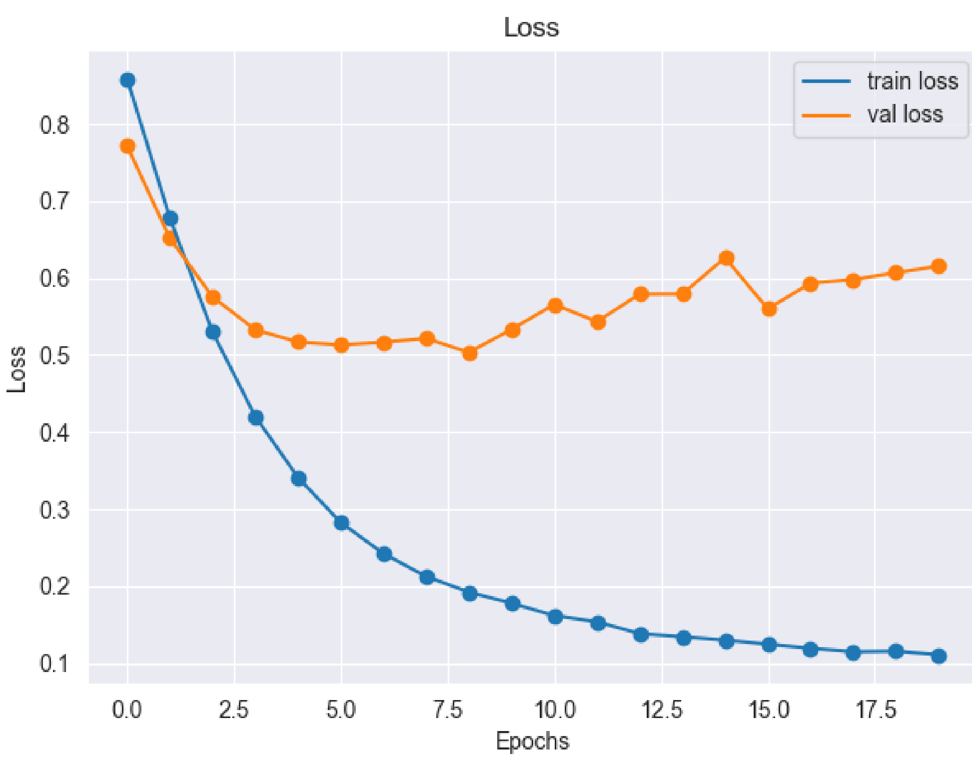
✓ 0.0s

```
sentimentBiLSTM(
  (embedding): Embedding(23571, 300)
  (lstm): LSTM(300, 64, batch_first=True, bidirectional=True)
  (fc): Linear(in_features=128, out_features=3, bias=True)
)
```

```python
torch.manual_seed(42)
torch.cuda.manual_seed(42)

# 实例化优化器
optimizer = optim.Adam(model.parameters())

criterion = nn.CrossEntropyLoss()

def acc(pred,label):
    pred = pred.argmax(1)
    return torch.sum(pred == label.squeeze()).item()
```

✓ 0.7s


Accuracy
train accuracy
val accuracy
accuracy
Epochs
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0.60
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5


Loss
train loss
val loss
Loss
Epochs
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5

```
Epoch 20
train_loss : 0.11079530822597278 val_loss : 0.6155028430124124
train_accuracy : 95.23585234805665 val_accuracy : 84.5719070546368
============================================================ >
```

4. Understand Word2Vec Code.

This code demonstrates the basics of training a Word2Vec model and using it to analyze word similarities. Here's a breakdown in 10 steps, explaining the core parts:

Step 1: Import Libraries

Import the required libraries:

gensim.models.Word2Vec: For training the Word2Vec model.

nltk.tokenize.word_tokenize: For tokenizing sentences into words.

sklearn.metrics.pairwise.cosine_similarity: For calculating cosine similarity between word vectors.

Step 2: Define Sentences

Define a list of sentences that will be used to train the Word2Vec model.

```
sentences = ["treasure today's day, as tomorrow is not promised.",
            "no matter how hard yesterday was, you can always start afresh today."]
```

Step 3: Tokenize Sentences

Use word_tokenize() from NLTK to split each sentence into a list of words (tokens).

```
tokenized_sentences = [word_tokenize(sentence) for sentence in sentences]
```

Step 4: Train the Word2Vec Model

Create a Word2Vec model instance and train it on the tokenized sentences. Key parameters include:

vector_size: Dimensionality of the word vectors (100 in this case).

window: Context window size (5 words before and after the target word).

min_count: Ignore words with frequency less than this value (1 here).

workers: Number of threads to use for training.

```
model = Word2Vec(tokenized_sentences, vector_size=100, window=5, min_count=1, workers=4)
```

Step 5: Retrieve Word Vectors

Access the trained word vectors from the model's vocabulary (model.wv) using the word as the key.

```
vector1 = model.wv['today']
vector2 = model.wv['yesterday']
vector3 = model.wv['afresh']
vector4 = model.wv['treasure']
```

Step 6: Examine Vector Length

Print the length of one of the word vectors to verify the dimensionality.

```
print(len(vector1))
```

Step 7: Calculate Cosine Similarity

Use cosine_similarity() from scikit-learn to compute the cosine similarity between pairs of word vectors. Cosine similarity measures the angle between vectors, indicating semantic relatedness.

```
print(cosine_similarity([vector1], [vector2]))
print(cosine_similarity([vector1], [vector3]))
print(cosine_similarity([vector1], [vector4]))
```

Step 8: Interpret Similarities

The cosine similarity values range from -1 to 1:

1: Vectors are identical, indicating high semantic similarity.

0: Vectors are orthogonal, indicating no relationship.

-1: Vectors point in opposite directions, indicating opposite meanings.

Step 9: Analyze Results

Based on the calculated cosine similarities, you can draw conclusions about the semantic relationships between the chosen words. For example, higher similarity scores suggest stronger relationships.

```python
from gensim.models import Word2Vec
from nltk.tokenize import word_tokenize
import nltk
from sklearn.metrics.pairwise import cosine_similarity


# Define sentences
sentences = ["treasure today's day, as tomorrow is not promised.",
             "no matter how hard yesterday was, you can always start afresh today."]

# Tokenize sentences
tokenized_sentences = [word_tokenize(sentence) for sentence in sentences]

# Train Word2Vec model
model = Word2Vec(tokenized_sentences, vector_size=100, window=5, min_count=1, workers=4)

# Retrieve vectors for specific words
vector1 = model.wv['today']
vector2 = model.wv['yesterday']
vector3 = model.wv['afresh']
vector4 = model.wv['treasure']

# Print length of a vector
print(len(vector1))

# Calculate and print cosine similarities
print(cosine_similarity([vector1], [vector2]))
print(cosine_similarity([vector1], [vector3]))
print(cosine_similarity([vector1], [vector4]))
```

[4] Python

```
100
[[0.02232039]]
[[0.00851715]]
[[-0.07085276]]
```

5. Understand Glove Code.

This code demonstrates how to use pre-trained GloVe word embeddings to analyze word similarities. Here's a breakdown in 10 steps, explaining the core parts:

Step 1: Import Libraries

Import the necessary libraries:

gensim.downloader: To download pre-trained word embedding models.

nltk.tokenize.word_tokenize: To tokenize sentences into words.

sklearn.metrics.pairwise.cosine_similarity: To calculate cosine similarity between word vectors.

Step 2: Download GloVe Embeddings

Use api.load() from Gensim to download the "glove-wiki-gigaword-100" pre-trained GloVe model. This model contains word vectors trained on a massive Wikipedia and Gigaword corpus.

```
glove_model = api.load("glove-wiki-gigaword-100")
```

Step 3: Define Sentences

Define a list of sentences containing the words you want to analyze.

```
sentences = ["treasure today's day, as tomorrow is not promised.",
             "no matter how hard yesterday was, you can always start afresh today."]
```

Step 4: Tokenize Sentences

Tokenize the sentences into individual words using word_tokenize() from NLTK.

```
tokenized_sentences = [word_tokenize(sentence) for sentence in sentences]
```

Step 5: Retrieve Word Vectors

Retrieve pre-trained word vectors from the glove_model using the get_vector() method.

```
vector1 = glove_model.get_vector('today')

vector2 = glove_model.get_vector('yesterday')

vector3 = glove_model.get_vector('afresh')

vector4 = glove_model.get_vector('treasure')
```

Step 6: Examine Vector Length

Print the length (dimensionality) of one of the word vectors to verify it matches the GloVe model's specification (100 in this case).

```
print(len(vector1))
```

Step 7: Calculate Cosine Similarity

Calculate the cosine similarity between pairs of word vectors using cosine_similarity() from scikit-learn.

```
print(cosine_similarity([vector1], [vector2]))

print(cosine_similarity([vector1], [vector3]))

print(cosine_similarity([vector1], [vector4]))
```

Step 8: Interpret Similarities

Analyze the cosine similarity values:

Higher values (closer to 1) indicate stronger semantic relationships between words.

Lower values (closer to 0) suggest less relatedness.

Negative values (closer to -1) imply opposite meanings.

```python
import gensim.downloader as api
from nltk.tokenize import word_tokenize
import nltk
from sklearn.metrics.pairwise import cosine_similarity

# Download GloVe vectors
glove_model = api.load("glove-wiki-gigaword-100")

# Define sentences
sentences = ["treasure today's day, as tomorrow is not promised.",
             "no matter how hard yesterday was, you can always start afresh today."]

# Tokenize sentences
tokenized_sentences = [word_tokenize(sentence) for sentence in sentences]

# Retrieve vectors for specific words
vector1 = glove_model.get_vector('today')
vector2 = glove_model.get_vector('yesterday')
vector3 = glove_model.get_vector('afresh')
vector4 = glove_model.get_vector('treasure')

# Print length of a vector
print(len(vector1))

# Calculate and print cosine similarities
print(cosine_similarity([vector1], [vector2]))
print(cosine_similarity([vector1], [vector3]))
print(cosine_similarity([vector1], [vector4]))
```

✓ 16.4s Python

```
100
[[0.7439699]]
[[0.09779004]]
[[0.2528864]]
```

Report score: ___________

Instructor's signature: ___________