电 子 科 技 大 学

UNIVERSITY OF ELECTRONIC SCIENCE AND TECHNOLOGY OF CHINA

实验报告

EXPERIMENT REPORT

|  |
| --- |
| C:\Users\ADMIN\AppData\Local\Temp\ksohtml20256\wps1.png |
| |  |  | | --- | --- | | **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.

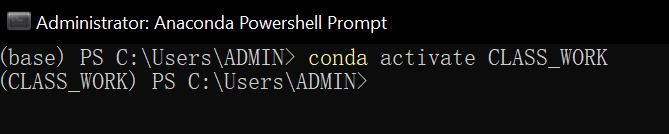
1. Experimental contents & step
2. using Jupyter for Natural Language Processing Tasks.
3. understand twitter\_Logistic.ipynb code.
4. understand bi\_lstm.ipynb code.
5. understand Word2Vec code.
6. understand GloVe code.
7. Experimental analysis

# 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:

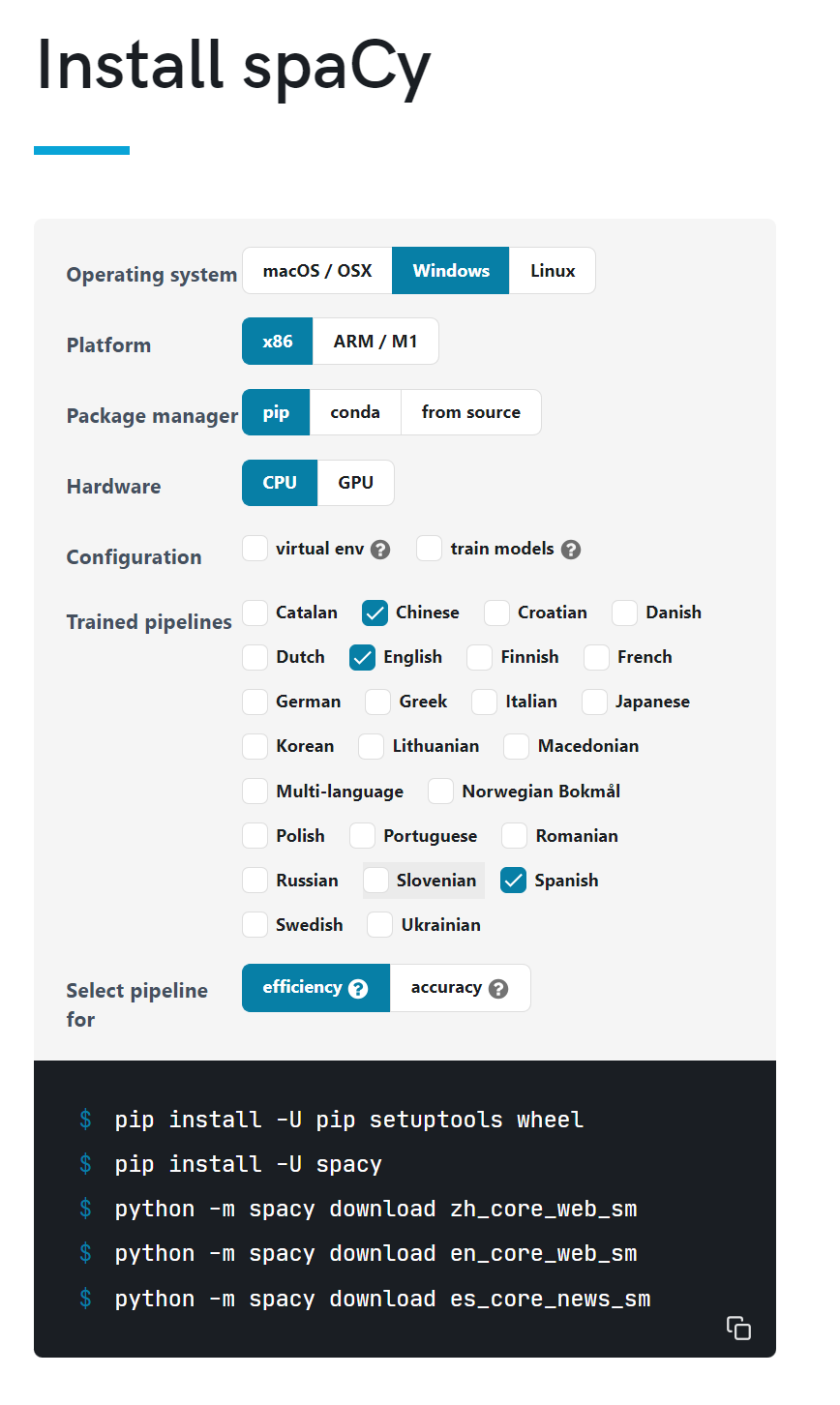
pip install -U pip setuptools wheel

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.

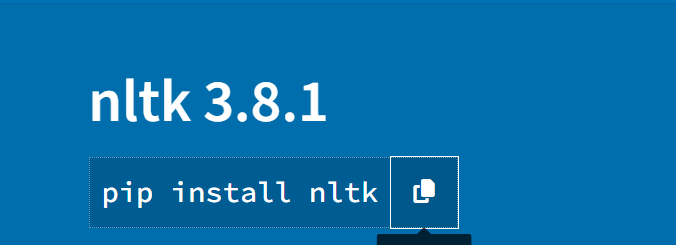




**Step 4: Install NLTK and Data**

Next, I installed the NLTK library and its data:

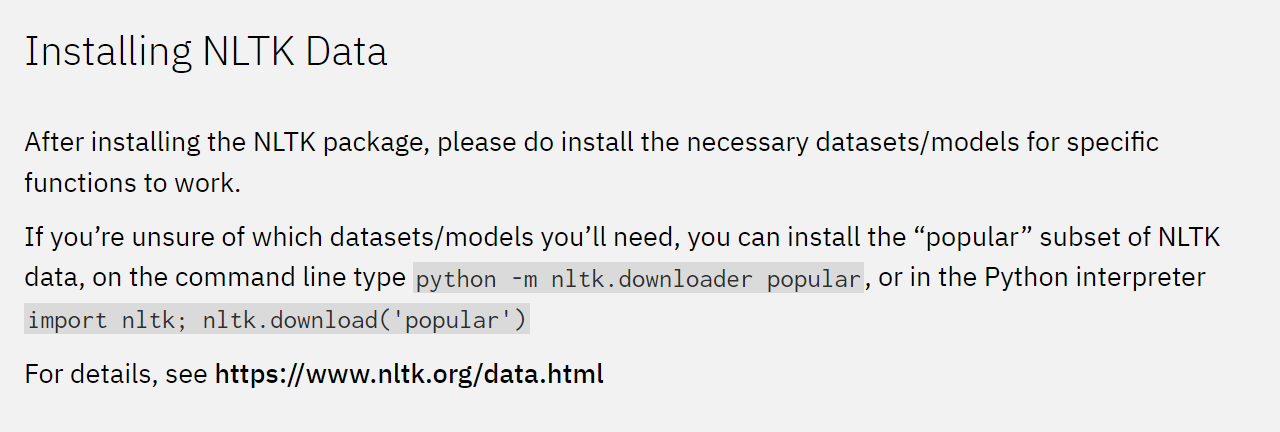
pip install nltk



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

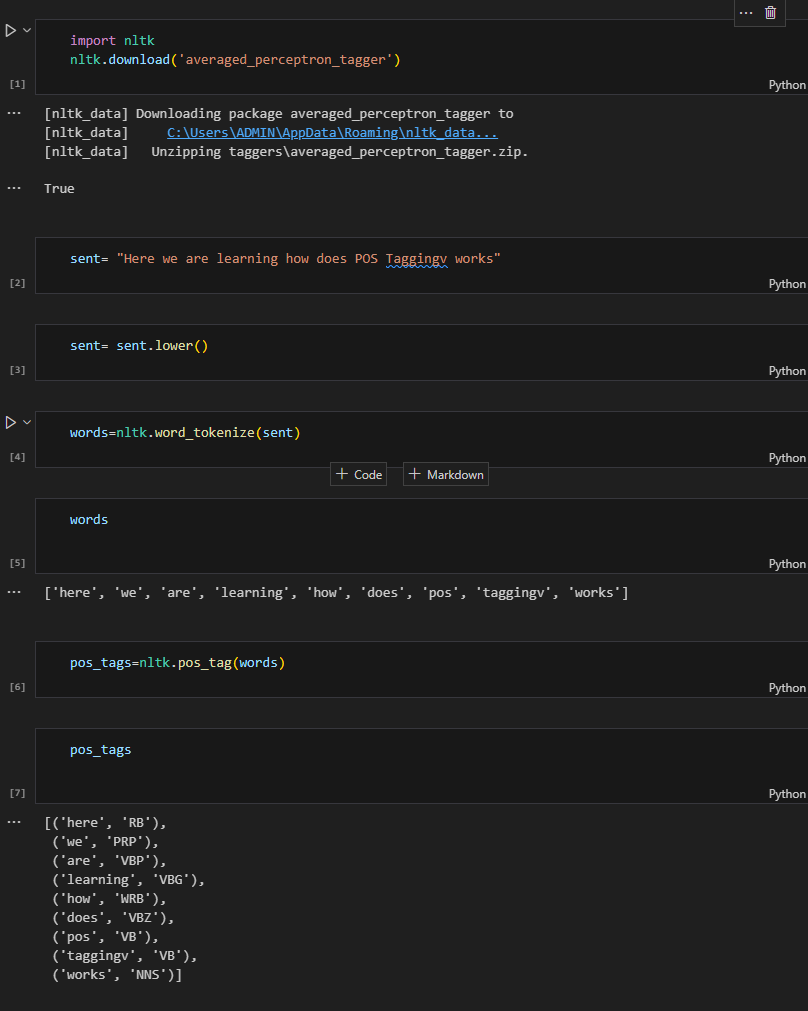
import nltk

nltk.download()



**Step 5: Experiment with NLTK**

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



**Step 6: Install TextBlob**

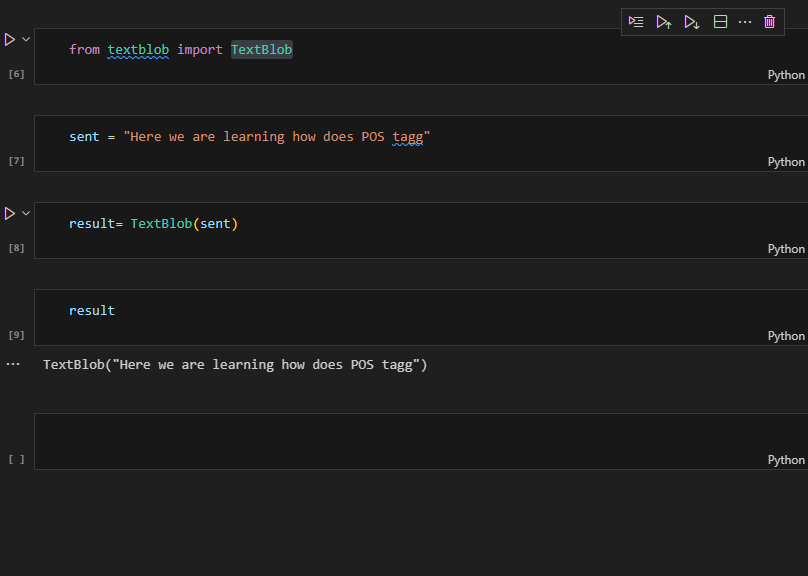
I installed TextBlob, another popular NLP library:

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:**

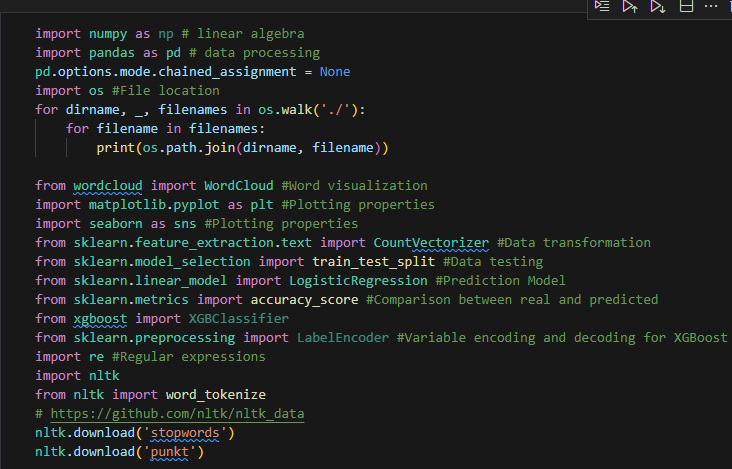
By leveraging Jupyter Notebook, I was able to seamlessly install and utilize various NLP libraries and language models. I conducted experiments with SpaCy, NLTK, and TextBlob, exploring their functionalities and comparing their performance on tasks like part-of-speech tagging. This interactive environment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

# 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.



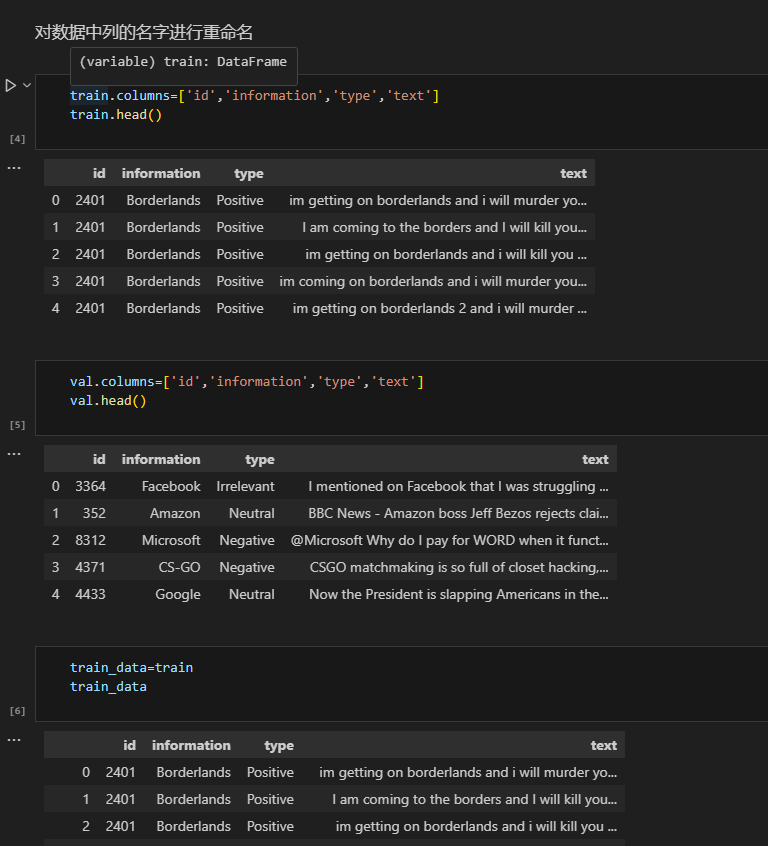
**Step 2: Load Datasets**

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.



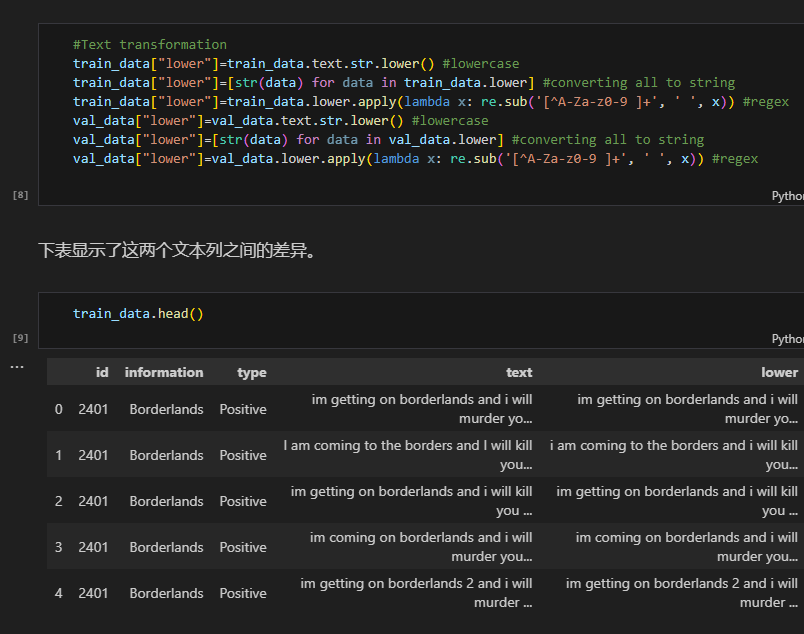
**Step 3: Rename Columns and Data Overview**

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.



**Step 4: Text Preprocessing**

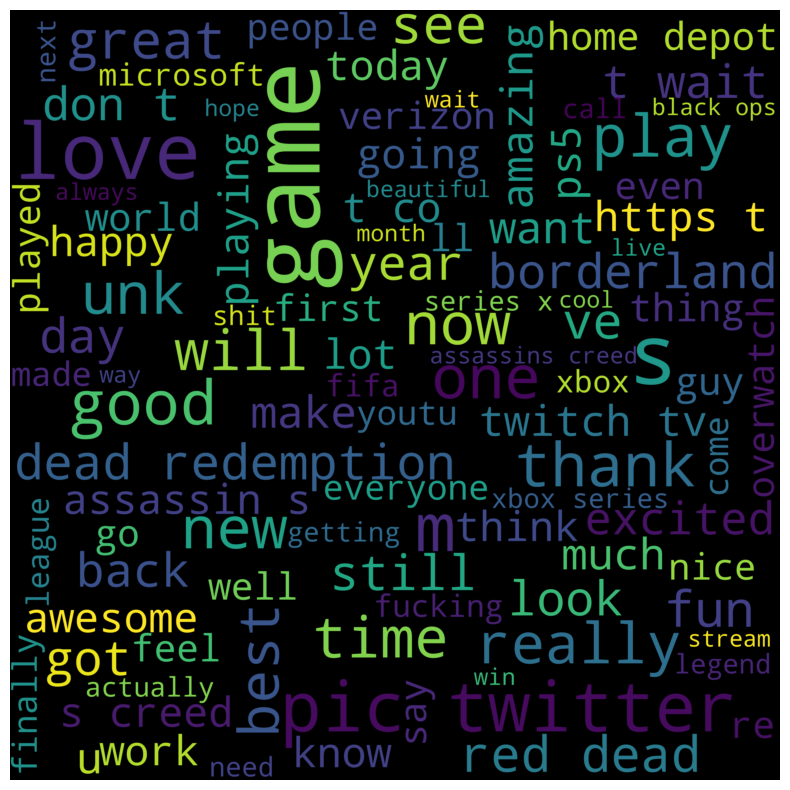
The tweet text is preprocessed to prepare it for analysis. This includes converting text to lowercase using str.lower(), ensuring all entries are strings, and removing special characters using regular expressions (re.sub()).



**Step 5: Feature Visualization**

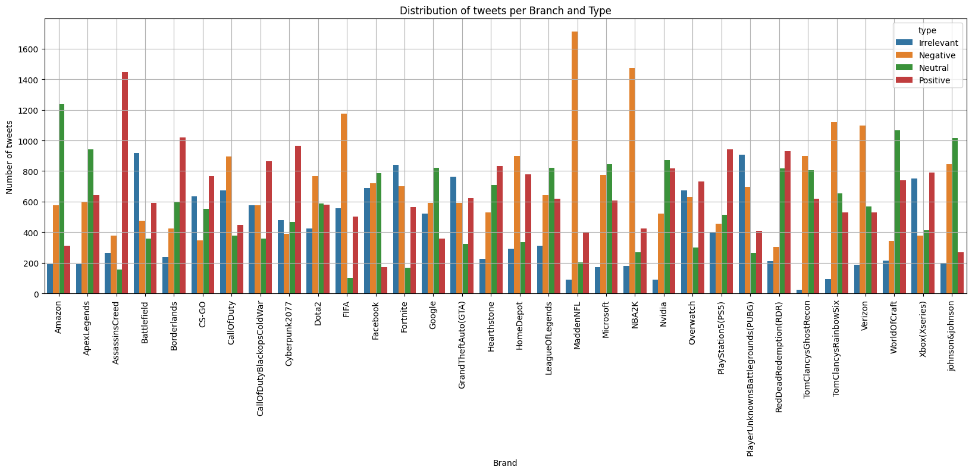
Word clouds are generated using the WordCloud module to visualize the most frequent words associated with each sentiment category ("Positive", "Negative", "Irrelevant", "Neutral").





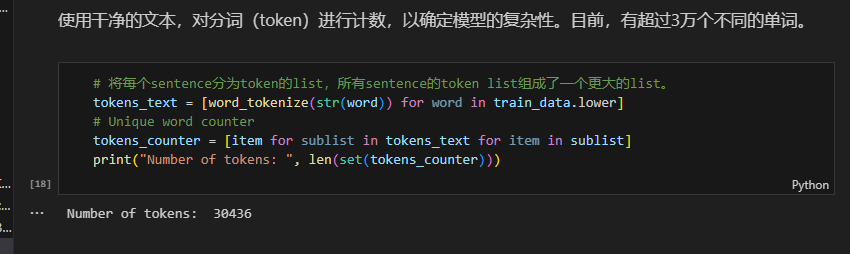
Step 6**: Data Distribution Visualization**

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



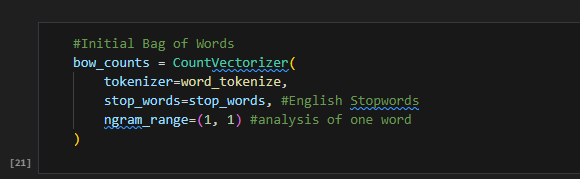
**Step 7:** **Tokenization and Stop Word Removal**

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.



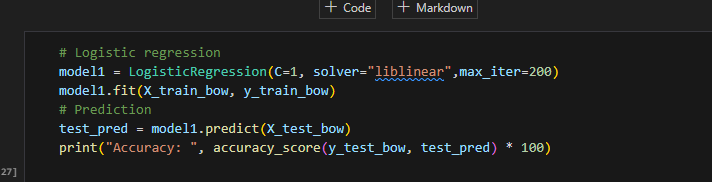
Step **8: Building the Bag-of-Words Model**

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.



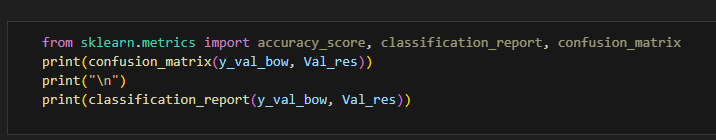
**Step 9:** **Training and Evaluating the Logistic Regression Model**

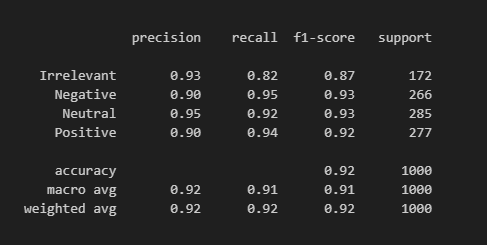
The data is split into training and testing sets using train\_test\_split(). The BoW model is fit to the training data using fit\_transform(), and the test data is transformed using the same vocabulary. A Logistic Regression model (LogisticRegression) is trained on the BoW features and evaluated on the test set using accuracy\_score().



**Step 10:** **Validating the Model**

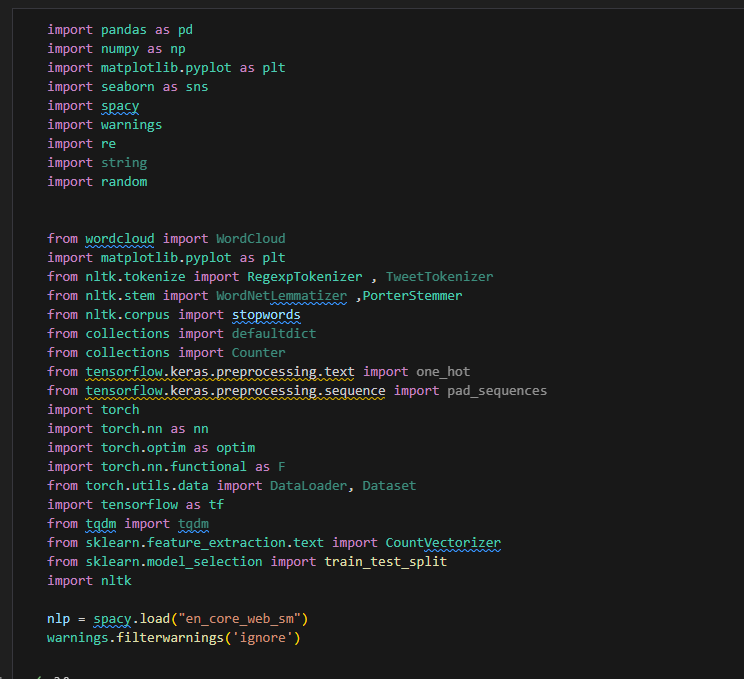
The trained Logistic Regression model is applied to the validation dataset to assess its performance on unseen data. The accuracy on the validation set is calculated, and a confusion matrix and classification report are generated using functions from sklearn.metrics to provide a detailed.





# Understand Bi\_Lstm.Ipynb Code.

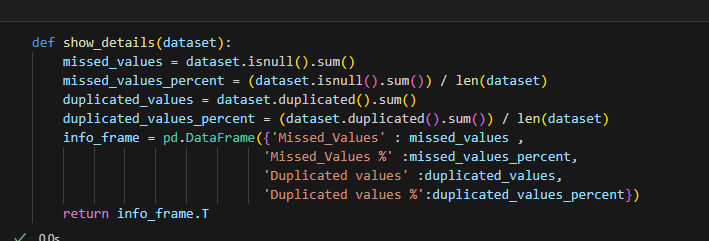
**tep 1:** Import Libraries: Import necessary libraries for data manipulation, visualization, text processing, and machine learning, including Pandas, NumPy, Matplotlib, Seaborn, SpaCy, NLTK, TensorFlow, PyTorch, and others.

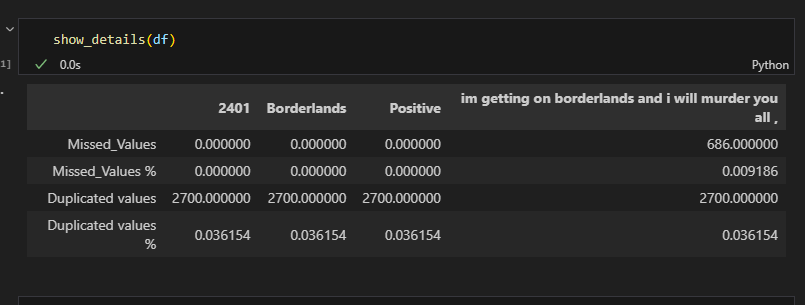


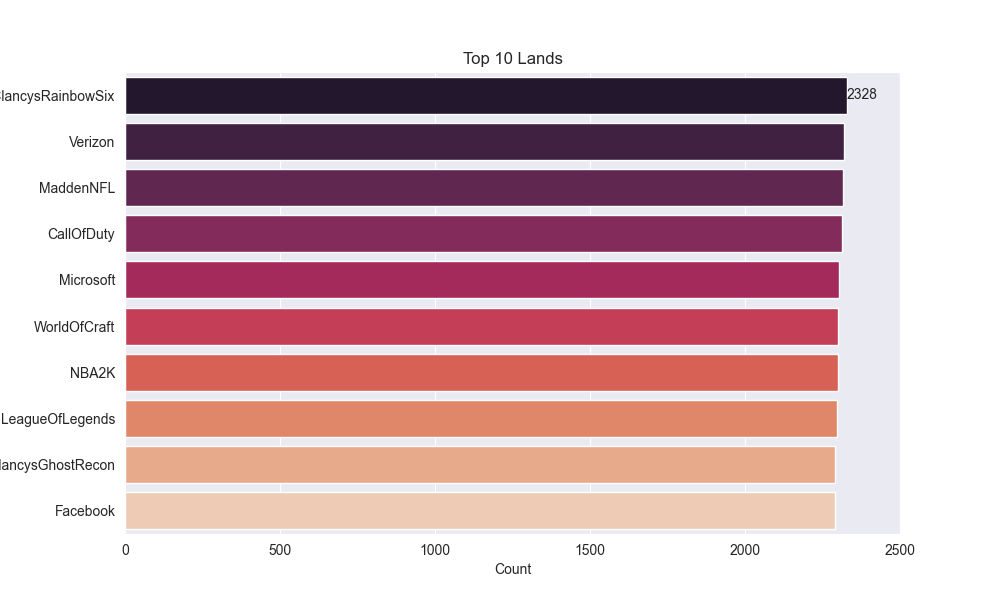
**Step 2:** Load and Inspect Dataset: Load the Twitter sentiment analysis dataset using pd.read\_csv(), examine its shape, column names, and data types, and display a few random samples.

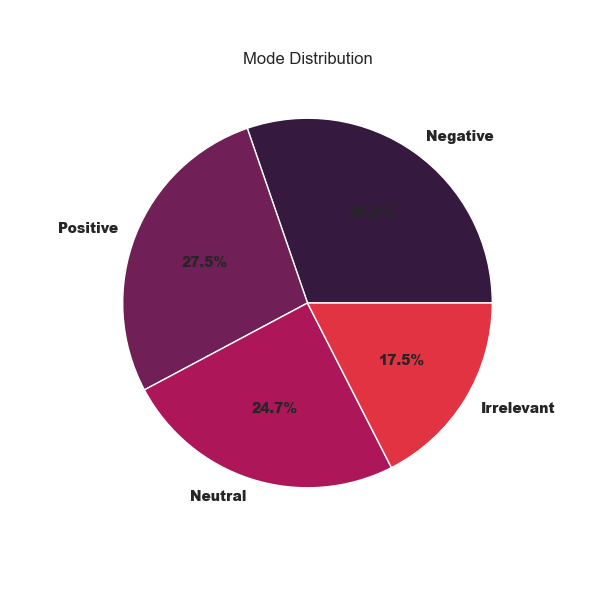


**Step 3:** Exploratory Data Analysis (EDA): Analyze the dataset for missing values, duplicates, and the distribution of data across different features like "Land" (brand) and "Mode" (sentiment). Visualize these distributions using bar plots and pie charts.





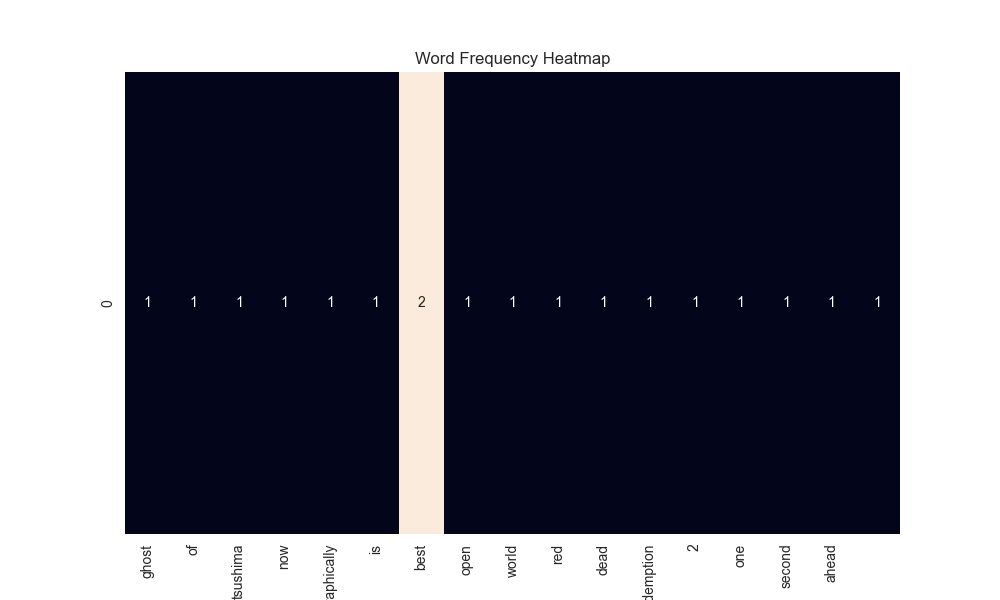
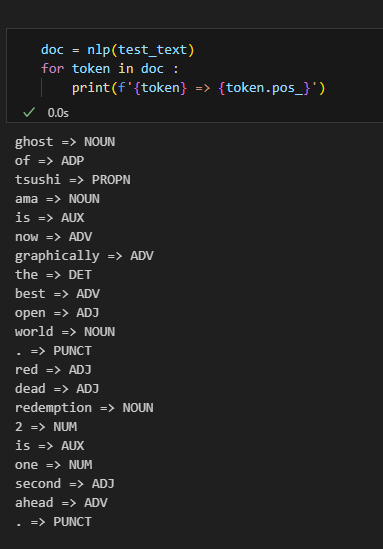


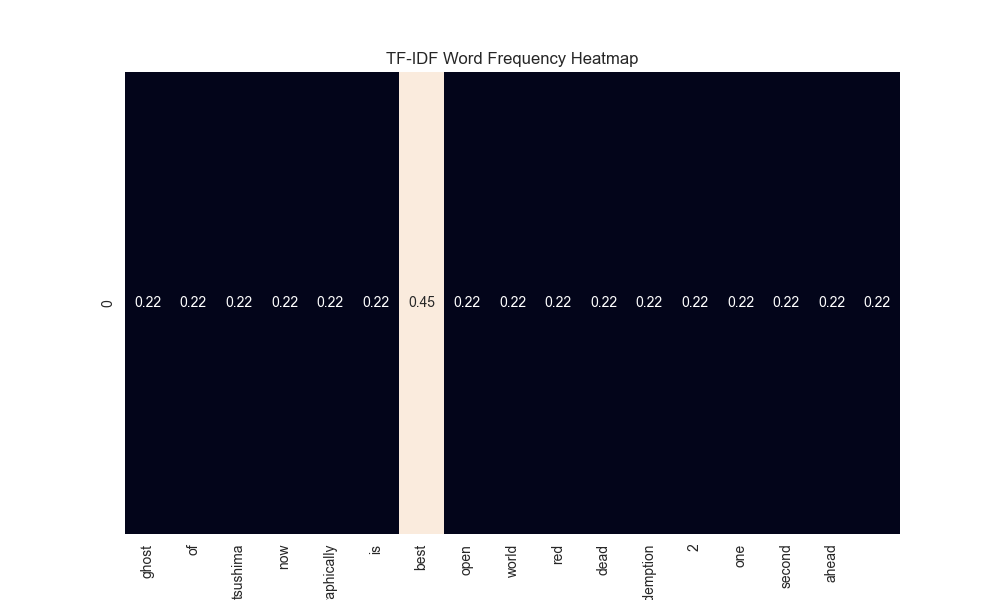


**Step 4:** Text Cleaning and Preprocessing: Apply custom functions (clean\_emoji() and text\_cleaner()) to clean the tweet text, including removing emojis, correcting common contractions, removing URLs and non-alphanumeric characters, and converting text to lowercase.

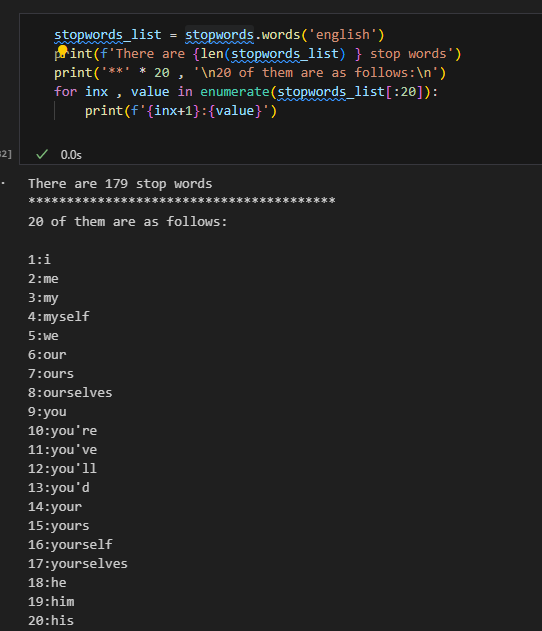


**Step 5:** Illust**rat**e Common NLP Techniques: Demonstrate various NLP concepts like part-of-speech tagging, named entity recognition, chunking, tokenization, counter vectorization, TF-IDF, and N-grams using examples from the dataset and relevant libraries like SpaCy and NLTK.



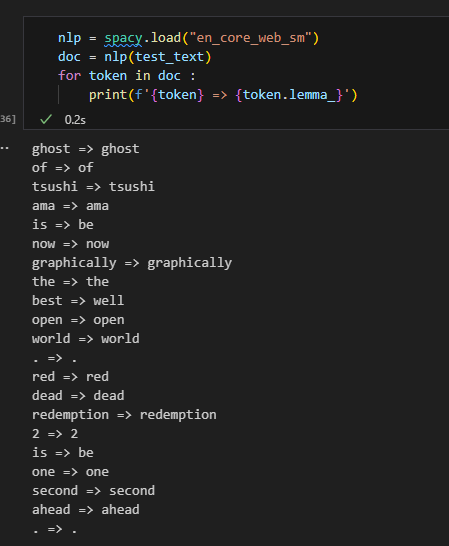


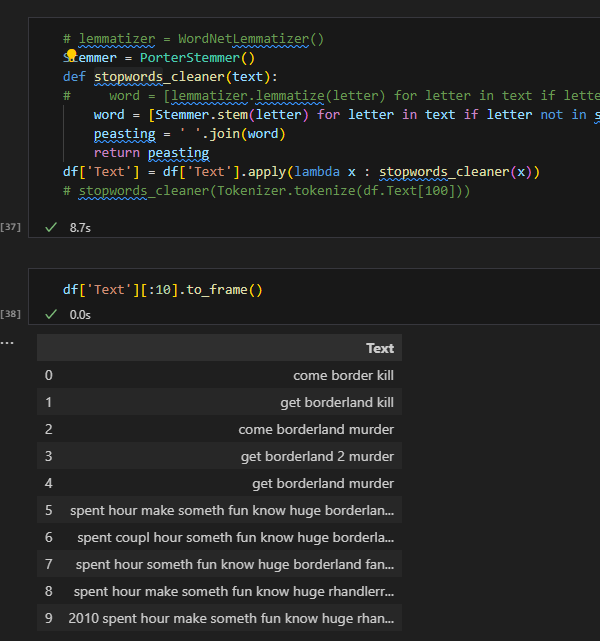
**Step 6:** Stop Word Analysis and Removal: Analyze the distribution of stop words across different sentiment categories and visualize the most frequent stop words using bar plots. Apply stop word removal to the tweet text using the stopwords\_cleaner() function and NLTK's list of stop words.



|  |  |
| --- | --- |
|  |  |
|  |  |

**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.

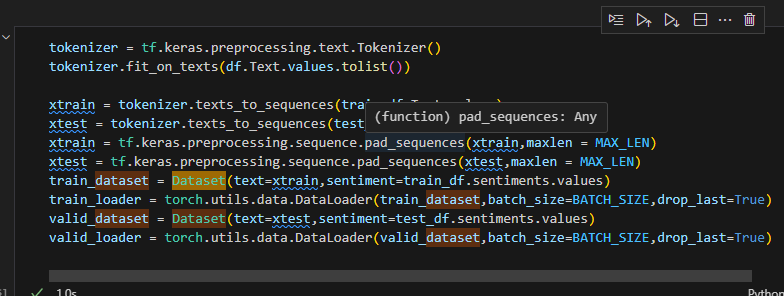




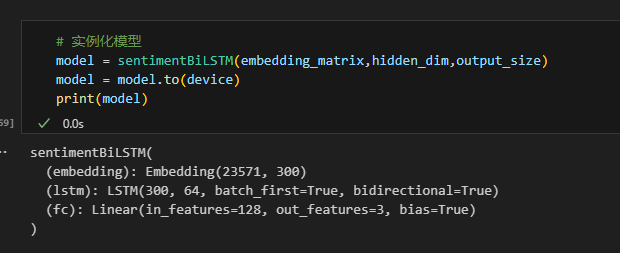
**Step 8:** Word Cloud Visualization: Generate word clouds for each sentiment category using the WordCloud module, providing a visual representation of the most frequent words associated with each sentiment.

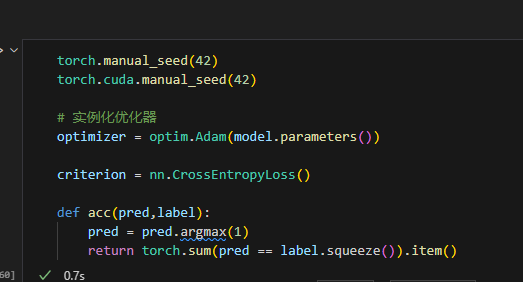
|  |  |
| --- | --- |
|  |  |
|  |  |

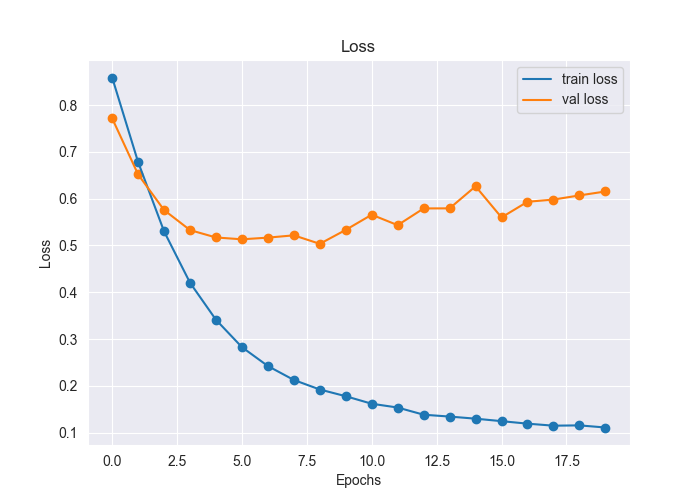
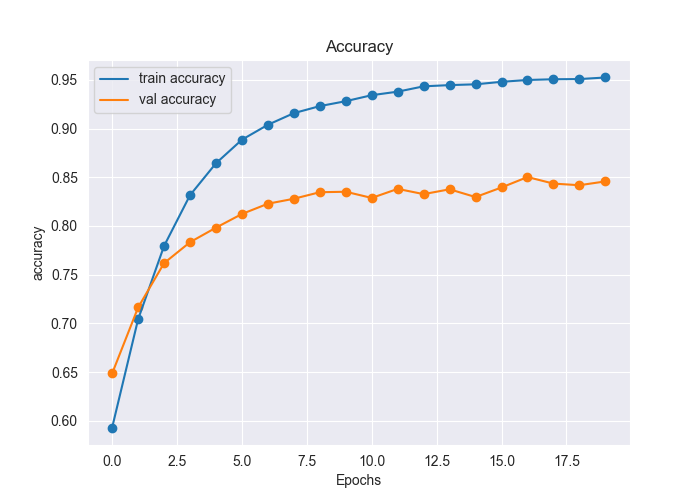
**Step 9:** Prepare Dataset for Deep Learning: Analyze the distribution of tweet lengths and preprocess the text data for input into a Bi-LSTM model. This includes converting text to sequences using Tokenizer.texts\_to\_sequences(), padding sequences to a fixed length (MAX\_LEN), and creating data loaders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

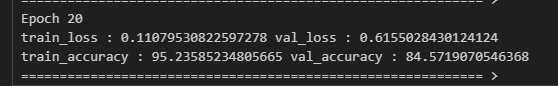


**Step 10:** Build, Train, and Evaluate the Bi-LSTM Model: Define a Bi-LSTM model architecture (sentimentBiLSTM) using PyTorch. Load pre-trained word embeddings (GloVe), train the model using an Adam optimizer and cross-entropy loss, and evaluate its performance on training and validation sets. Visualize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









# 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:

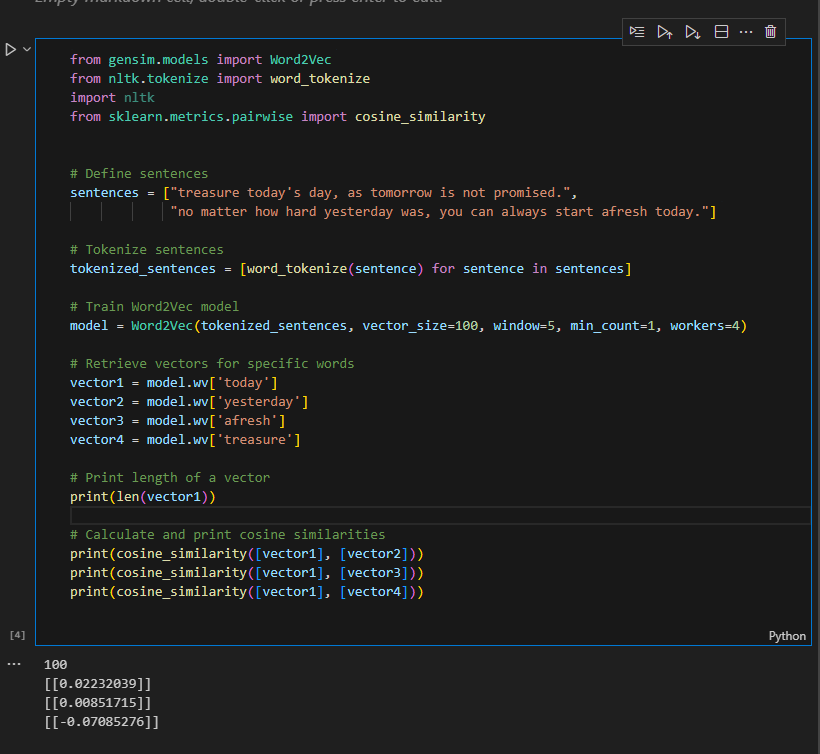
1: Vectors are identical, indicating high semantic similarity.

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.



# 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]))

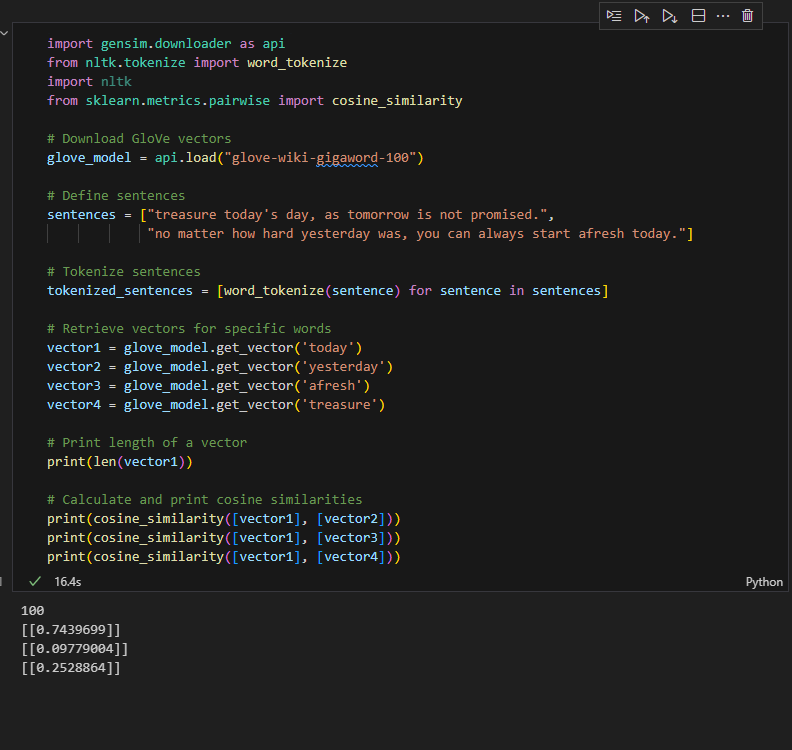
**Step 8: Interpret Similarities**

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.



Report score: .

Instructor's signature: .