BANSILAL RAMNATH AGARWAL CHARITABLE TRUST'S

VISHWAKARMA INSTITUTE OF INFORMATION TECHNOLOGY

PUNE – 411018

College of Engineering

DEPARTMENT OF AI&DS ENGINEERING

INDEX

This is to Certify that Mr. Sanket fulzele

Of class TY (A) Roll no. 371018 Exam seat no. 22110728 has performed the abovementioned 9 Number of experiment in the NLP Subject laboratory in the Departmentof AI&DS Engineering at Vishwakarma Institute of Information Technology Pune.

Date_____ In charge Faculty Head of Department

Name: Sanket S. Fulzele

Roll No: 371018

PRN: 22110728

Div: A

Objective:

Comparative study of available libraries for Natural Language processing with respect to provided functionalities, platform dependence, supported NLP approaches, supported NLP Tasks, advantages and Disadvantages etc.

Library Selection:

1. NLTK (Natural Language Toolkit):

- Features: NLTK is a comprehensive library that provides tools for various NLP tasks, including tokenization, stemming, tagging, parsing, and more.
- Use Case: It is often used for educational purposes, research, and prototyping due to its extensive functionalities.

2. spaCy:

- Features: spaCy is known for its speed and efficiency. It offers pre-trained models for various languages and is suitable for production environments. It provides tools for tokenization, named entity recognition, part-of-speech tagging, and more.
- Use Case: SpaCy is commonly used in production environments where speed and accuracy are crucial.

3. Gensim:

- Features: Gensim is primarily used for topic modelling and document similarity analysis. It is efficient in handling large text corpora and offers implementations of algorithms like Word2Vec.
- Use Case: Gensim is often used in research and applications that involve analysing large text datasets for topic modelling.

4. Transformers (Hugging Face):

- Features: The Transformers library by Hugging Face provides pre-trained models for a wide range of NLP tasks, including text classification, named entity recognition, and language translation.
- Use Case: It is widely used for state-of-the-art results in various NLP applications by leveraging pre-trained transformer-based models like BERT, GPT, etc.

5. Text Blob:

- Features: Text Blob is a simple and easy-to-use library for common NLP tasks. It wraps NLTK's functionality with a simplified API.
- Use Case: It is suitable for beginners and small-scale projects that require basic NLP functionalities.

6. Stanford NLP:

- Features: The Stanford NLP library provides tools for part-of-speech tagging, named entity recognition, sentiment analysis, and more. It is implemented in Java but has wrappers for other languages.
- Use Case: It is often used in research and projects requiring robust NLP capabilities.

2. Criteria for Comparison:

Criterion	spaCy	NLTK	Gensim
Ease of Use	Beginner-friendly, pre-trained models	Steeper learning curve, fine-grained control	Focuses on specific tasks, integrates with others
Processing Speed	Highly efficient for common tasks	Varies by task and algorithm	Efficient for topic modeling
Community Support	Growing community, good resources	Large and active community, extensive resources	Smaller community, active discussions
Available Functionalities	Core NLP tasks with pre-trained models	Extensive range, including stemming and sentiment analysis	Primarily topic modeling and word vectors
Languages Supported	Primarily English, some pre-trained models for other languages	Primarily English, additional language modules	Language agnostic
Dependencies	NumPy, spaCy-specific libraries for models	NumPy, nltk packages for specific functionalities	NumPy, SciPy (optional)
Strengths	Pre-trained models, efficiency, production-ready	Flexibility, control, research	Topic modeling, document similarity, word vectors

Weaknesses	Less flexibility, smaller community	Steeper learning curve, slower for some tasks	Limited to specific tasks
Best for	Beginners, prototyping, production applications	Research, custom tasks, teaching	Topic modeling, large text analysis, recommender systems

Task	spaCy	NLTK	Gensim
Tokenization	Pre-trained models	Various modules	Not directly
Part-of-Speech Tagging	Pre-trained models	Various taggers	N/A
Named Entity Recognition	Pre-trained models	Requires training data	N/A
Dependency Parsing	Pre-trained models	Statistical parsers	N/A
Topic Modeling	N/A	Not dedicated, text processing tools	Highly efficient, specialized algorithms
Document Similarity	N/A	Various distance measures	Word vectors

Performance:

- Spacy Generally Faster For Core Tasks.
- Accuracy Similar For All Libraries With Pre-Trained Models, Fine-Tuning Can Improve.
- Spacy And Gensim More Memory-Efficient For Large Datasets.

Community and Documentation:

- NLTK has the largest and most active community.
- spaCy has a rapidly growing community with good resources.
- Gensim has a smaller but dedicated community with good resources.

Dependencies:

- All Libraries Require Python And NumPy.
- Spacy Needs Specific Libraries For Different Models.
- NLTK Requires Additional Packages For Specific Functionalities.
- Gensim May Require SciPy For Some Algorithms.

Conclusion:

The choice of an NLP library depends on your project objectives and expertise. SpaCy is well-suited for beginners and production environments due to its fast performance and pre-trained models. NLTK, with its flexibility and extensive functionalities, is ideal for research and customization tasks. Gensim excels in analyzing large text and topics, offering efficient algorithms and memory-friendly operations. Select the library that aligns with your language requirements, whether it's the availability of pre-trained models in specific languages with spaCy, the modularity of NLTK, or Genism’s language-agnostic approach.

Name: Sanket S. Fulzele

Roll No: 371018

PRN: 22110728

Div: A

Objective:

Perform various pre-processing tasks like tokenization, stemming,lemmatization, stop word removal.

NLP preprocessing Steps

In natural language processing (NLP), preprocessing refers to the tasks and techniques used to prepare text data for analysis and modelling. Here's an introduction to various preprocessing tasks commonly used in NLP:

1. Tokenization: Breaking text into smaller units, typically words or tokens. This step is crucial for further analysis as it allows the algorithm to work with individual elements of the text.
2. Lowercasing: Converting all text to lowercase. This helps in standardizing the text and treating words with different cases as the same, reducing the vocabulary size.
3. Stop word Removal: Removing common words like "and," "the," "is," etc., which do not carry significant meaning and can be noise in the analysis.
4. Punctuation Removal: Eliminating punctuation marks from the text, as they often do not contribute to the semantics of the text and can be distracting for models.
5. Normalization: Converting words to their base or root form. This includes tasks like stemming (reducing words to their base form by removing suffixes) and lemmatization (reducing words to their base form based on the dictionary).
6. Spell Checking and Correction: Correcting spelling errors in the text using algorithms that compare words against a known dictionary or language model.
7. Token Filtering: Removing tokens based on specific criteria, such as removing numbers, special characters, or specific types of words based on domain knowledge.
8. Part-of-Speech (POS) Tagging: Assigning grammatical tags to each word in the text, such as noun, verb, adjective, etc. This information is useful for tasks like named entity recognition and syntactic analysis.
9. Entity Recognition: Identifying and categorizing entities like names of people, organizations, locations, etc., in the text.
10. Vectorization: Converting text data into numerical vectors, often using techniques like Bag-of-Words (BoW), Term Frequency-Inverse Document Frequency (TF-IDF), or word embeddings like Word2Vec or GloVe. This step is essential for machine learning models to process text data. These preprocessing tasks are often applied in combination, depending on the specific NLP task and the characteristics of the text data being analysed

```
!pip install indic-nlp-library
```

```
!pip install spacy
```

```
!pip install  pyPDF2
```

```
import nltk
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer, PorterStemmer
from nltk.probability import FreqDist
from nltk.corpus import stopwords
from PyPDF2 import PdfReader as pdf
```

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

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Unzipping tokenizers/punkt.zip.
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Unzipping corpora/stopwords.zip.
True
```

```
def extract_text_from_pdf(pdf_path):
    text = ""
    with open(pdf_path, 'rb') as file:
        reader = pdf(file)
        num_pages = len(reader.pages)
        for page_num in range(num_pages):
            page = reader.pages[page_num]
            text += page.extract_text()
    return text
```

```
pdf_path = "/content/drive/MyDrive/PHY 1.pdf"
corpus=extract_text_from_pdf(pdf_path)
```

```
# text = "Natural language processing (NLP) is a field of computer science, artificial intelligence (AI), and li

tokens = word_tokenize(corpus)
```

```
lemmatizer = WordNetLemmatizer()
lemmatized_tokens = [lemmatizer.lemmatize(token) for token in tokens]
```

```
stemmer = PorterStemmer()
stemmed_tokens = [stemmer.stem(token) for token in tokens]
```

```
# Remove stopwords
stop_words = set(stopwords.words('english'))
filtered_tokens = [token for token in tokens if token.lower() not in stop_words]
```

```
# Count total unique words
total_unique_words = len(set(tokens))
```

```
# Calculate Type-Token Ratio (TTR)
type_token_ratio = len(set(tokens)) / len(tokens)
```

```
# print("Original Text:")
# print(text)
print("\nTokenization:")
print(tokens)
print("\nLemmatization:")
print(lemmatized_tokens)
print("\nStemming:")
print(stemmed_tokens)
print("\nTotal Unique Words:", total_unique_words)
print("\nType-Token Ratio (TTR):", type_token_ratio)
```

```
Tokenization:
['PHYSICS', 'PBL', 'NAME', ':', 'SANKET', 'SANJAY', 'FULZELE', 'Div', '.', 'H', 'ROLL', 'No', ':', '842', 'B

Lemmatization:
['PHYSICS', 'PBL', 'NAME', ':', 'SANKET', 'SANJAY', 'FULZELE', 'Div', '.', 'H', 'ROLL', 'No', ':', '842', 'B

Stemming:
['physic', 'pbl', 'name', ':', 'sanket', 'sanjay', 'fulzel', 'div', '.', 'h', 'roll', 'no', ':', '842', 'bat

Total Unique Words: 487

Type-Token Ratio (TTR): 0.45471521942110177
```

```
!python -m spacy download es_core_news_sm
```

```
import spacy
from collections import Counter

nlp = spacy.load("es_core_news_sm")

text = "La rápida zorra marrón salta sobre el perro perezoso."

doc = nlp(text)
tokens_lemmas = [(token.text, token.lemma_) for token in doc]


from nltk.stem import SnowballStemmer
stemmer = SnowballStemmer(language='spanish')
stemmed_tokens = [stemmer.stem(token.text) for token in doc]

unique_words = set(token.text for token in doc)

type_token_ratio = len(unique_words) / len(tokens_lemmas)

print("Tokenization and Lemmatization:")
print(tokens_lemmas)
print("\nStemming:")
print(stemmed_tokens)
print("\nTotal Unique Words:", len(unique_words))
print("Type Token Ratio:", type_token_ratio)
```

```
Tokenization and Lemmatization:
[('La', 'el'), ('rápida', 'rápido'), ('zorra', 'zorra'), ('marrón', 'marrón'), ('salta', 'salta'), ('sobre',

Stemming:
['la', 'rap', 'zorr', 'marron', 'salt', 'sobr', 'el', 'perr', 'perez', '.']

Total Unique Words: 10
Type Token Ratio: 1.0
```

Name: Sanket S. Fulzele

Roll No: 371018

PRN: 22110728

Div: A

1. Introduction of Available Word Similarity Measures:

- Provide an overview of word similarity measures commonly used in Natural Language Processing (NLP).
- Discuss the importance of word similarity measures in tasks like spell checking, auto-correction, and information retrieval.
- Explain the concept of Minimum Edit Distance (also known as Levenshtein distance) and its significance in measuring the similarity between two words.

2. Manual Calculation of Minimum Edit Distance:

- (Saturday, Sunday): 3
- (Eaten, Beaten): 2
- (Minimum, Maximum): 3

3. Create a List of Lexicons for the Spelling Checker:

- Compile a list of lexicons or dictionaries for two languages of your choice.

- Ensure the lexicons cover a wide range of vocabulary and include commonly used words in the respective languages.
- Each entry in the lexicons should include the word's correct spelling and any additional information (e.g., part of speech, frequency of use).

4. Develop a Spelling Checker using Minimum Edit Distance:

- Implement a spelling checker using the Minimum Edit Distance algorithm for the two chosen languages.
- The spelling checker should take input text and identify misspelled words by comparing them to the words in the lexicons.
- Provide suggestions for correcting misspelled words based on the words in the lexicons that have the shortest edit distance.
- Ensure the spelling checker is user-friendly and capable of handling input text efficiently.

5. Conclusion:

- Summarize the key findings and outcomes of developing the spelling checker using Minimum Edit Distance.
- Reflect on the challenges encountered during the development process and discuss any insights gained.
- Offer recommendations for improving the spelling checker or extending its functionality in future iterations.

```python
import nltk

# Download NLTK word list (if not already downloaded)
nltk.download('words')

from nltk.corpus import words

def min_edit_distance(source, target):
    m = len(source)
    n = len(target)

    # Initialize a matrix to store the edit distances
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Initialize the first row and column
    for i in range(m + 1):
        dp[i][0] = i
    for j in range(n + 1):
        dp[0][j] = j

    # Fill in the matrix using dynamic programming
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            cost = 0 if source[i - 1] == target[j - 1] else 1
            dp[i][j] = min(dp[i - 1][j] + 1,      # Deletion
                           dp[i][j - 1] + 1,      # Insertion
                           dp[i - 1][j - 1] + cost)  # Substitution

    return dp[m][n]

def spelling_checker(word, dictionary):
    # Find the closest match in the dictionary
    min_distance = float('inf')
    closest_match = None

    for candidate in dictionary:
        distance = min_edit_distance(word, candidate)
        if distance < min_distance:
            min_distance = distance
            closest_match = candidate

    return closest_match, min_distance

# Take user input
word_to_check = input("Enter a word: ")

# Use NLTK words list as the dictionary
dictionary = words.words()

closest_word, min_distance = spelling_checker(word_to_check, dictionary)
print(f"Suggested correction for '{word_to_check}': {closest_word}")
print(f"Minimum edit distance: {min_distance}")
```

OUTPUT:

```
[nltk_data] Downloading package words to
[nltk_data]     C:\Users\Dell\AppData\Roaming\nltk_data...
[nltk_data]   Package words is already up-to-date!
Enter a word: helllo
Suggested correction for 'helllo': hello
Minimum edit distance: 1

Enter a word: summmeer
Suggested correction for 'summmeer': summer
Minimum edit distance: 2

Enter a word: eeneergy
Suggested correction for 'eeneergy': energy
Minimum edit distance: 2

Enter a word: Happinneess
Suggested correction for 'Happinneess': happiness
Minimum edit distance: 3

Enter a word: intelligence
Suggested correction for 'intelligence': intelligence
Minimum edit distance: 0
```

Name: Sanket S. Fulzele

Roll No: 371018

PRN: 22110728

Div: A

POS Tagging In Different Languanges

Part-of-Speech (PoS) tagging involves labeling words in a sentence with their grammatical categories, such as nouns, verbs, adjectives, and so on. Different languages have their own PoS tag sets tailored to their grammatical structures and linguistic nuances. Here's a brief overview of PoS tag sets for some languages:

1. **English**:
    - Universal PoS Tag Set (Universal Dependencies): This tag set includes tags like NOUN (noun), VERB (verb), ADJ (adjective), ADV (adverb), PRON (pronoun), DET (determiner), ADP (adposition), CONJ (conjunction), and more. It aims to provide a universal standard for PoS tagging across languages.
2. **Spanish**:
    - Universal Dependencies for Spanish: Similar to English, Spanish also follows the Universal PoS Tag Set with tags like NOUN, VERB, ADJ, ADV, PRON, DET, ADP, CONJ, etc. Additionally, Spanish has specific tags for reflexive verbs (REFL), gerunds (GRND), and other grammatical constructs.
3. **French**:
    - Universal Dependencies for French: French PoS tags include similar categories to English and Spanish but may have additional tags for gender and number agreement (e.g., NOUN.FEM for feminine nouns, NOUN.PL for plural nouns).
4. **German**:
    - Universal Dependencies for German: German PoS tags also align with the universal tag set but may include specific tags for case markings (e.g., NOUN.NOM for nominative case, NOUN.ACC for accusative case).

Types of POS Tagging

Part-of-Speech (PoS) tagging is a fundamental task in natural language processing (NLP) that involves labeling words in a text with their corresponding grammatical categories, such as nouns, verbs, adjectives, adverbs, etc. Several approaches are used for PoS tagging, each with its strengths and limitations. Here's a brief discussion of some common approaches:

1. **Rule-Based Tagging**:
    - **Dictionary Lookup**: This approach involves using a dictionary of words along with their associated PoS tags. Words are looked up in the dictionary to assign tags based on predefined rules.
    - **Rule-Based Patterns**: Rules and patterns are defined based on linguistic knowledge and regularities in language. For example, words ending in "-ing" are often verbs, and words starting with capital letters are likely proper nouns.
2. **Statistical Tagging**:
    - **N-gram Models**: These models use sequences of N words (e.g., bigrams, trigrams) to predict the PoS tag of a word based on the context provided by surrounding words.
    - **Hidden Markov Models (HMMs)**: HMMs model the probability of transitioning between PoS tags and the probability of observing words given their tags, learning from annotated training data.
    - **Maximum Entropy Models**: These models assign probabilities to PoS tags based on features extracted from the context, such as neighboring words, prefixes, suffixes, and word shapes.
3. **Machine Learning-Based Tagging**:
    - **Supervised Learning**: Utilizes annotated training data to train classifiers (e.g., decision trees, support vector machines, neural networks) to predict PoS tags based on features derived from the text.
    - **Semi-Supervised Learning**: Combines labeled and unlabeled data to improve tagging accuracy, often using techniques like self-training or co-training.

- **Deep Learning Models**: Neural network architectures such as recurrent neural networks (RNNs), long short-term memory (LSTM) networks, and transformer models (e.g., BERT) have shown state-of-the-art performance in PoS tagging by learning complex contextual representations.

4. **Hybrid Approaches**:
   - **Combining Rules and Statistics**: Hybrid models integrate rule-based mechanisms with statistical or machine learning components to benefit from both linguistic knowledge and data-driven learning.
   - **Ensemble Methods**: Combining predictions from multiple taggers, each using a different approach or model, can improve overall tagging accuracy and robustness.

Each approach has its advantages and is suited to different scenarios based on available data, computational resources, and the desired balance between linguistic accuracy and scalability. Modern NLP systems often employ a combination of these approaches to achieve high-performance PoS tagging.

1. Inbuilt PoS Tagging Using Libraries

```
import nltk
nltk.download("punkt")
nltk.download('averaged_perceptron_tagger')
from textblob import TextBlob

# Using NLTK
text = "The quick brown fox jumps over the lazy dog"
tokens = nltk.word_tokenize(text)
pos_tags_nltk = nltk.pos_tag(tokens)
print("NLTK PoS Tags:", pos_tags_nltk)

# Using TextBlob
blob = TextBlob(text)
pos_tags_textblob = blob.tags
print("TextBlob PoS Tags:", pos_tags_textblob)
```

```
NLTK PoS Tags: [('The', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'VBZ'), ('over', 'IN'), ('the', 'DT'), ('l
TextBlob PoS Tags: [('The', 'DT'), ('quick', 'JJ'), ('brown', 'NN'), ('fox', 'NN'), ('jumps', 'VBZ'), ('over', 'IN'), ('the', 'DT'),
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /root/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
```

2. Regular Expressions for PoS Categories

```
from nltk import RegexpTagger

patterns = [
    (r'.*ing$', 'VBG'),
    (r'.*ed$', 'VBD'),
    (r'.*es$', 'VBZ'),
    (r'.*ould$', 'MD'),
    (r'.*\'s$', 'NN$'),
    (r'.*s$', 'NNS'),
    (r'^-?[0-9]+(\.[0-9]+)?$', 'CD'),
    (r'the', 'DT'),
    (r'in', 'IN'),
    (r'.*', 'NN')  # Default: nouns
]

regexp_tagger = RegexpTagger(patterns)

# Example sentence
sentence = "5 friends have been singing in the rain"
tagged_words = regexp_tagger.tag(sentence.split())
print(tagged_words)
```

```
[('5', 'CD'), ('friends', 'NNS'), ('have', 'NN'), ('been', 'NN'), ('singing', 'VBG'), ('in', 'IN'), ('the', 'DT'), ('rain', 'NN')]
```

3. Dictionary/Lookup Table for PoS Tagging

```
pos_dict = {
    'dog': 'NN',        # Noun
    'jump': 'VB',       # Verb
    'quick': 'JJ',      # Adjective
    'lazy': 'JJ',       # Adjective
    'brown': 'JJ',      # Adjective
    'fox': 'NN',        # Noun
    'over': 'IN',       # Preposition
    'the': 'DT',        # Determiner
    'tree': 'NN',       # Noun
    'eats': 'VB',       # Verb
    'apple': 'NN',      # Noun
    'and': 'CC',        # Conjunction
    'orange': 'NN',     # Noun
    'juice': 'NN',      # Noun
    'big': 'JJ',        # Adjective
    'small': 'JJ',      # Adjective
    'quickly': 'RB',    # Adverb
    'quietly': 'RB',    # Adverb
    'very': 'RB',       # Adverb
    'eat': 'VB'         # Verb
}

def pos_tag_with_dict(sentence):
    tokens = nltk.word_tokenize(sentence)
    pos_tags = [(token, pos_dict.get(token.lower(), 'NN')) for token in tokens]
    return pos_tags

# Example usage:
text_to_tag = "The quick brown fox jumps over the lazy dog"
pos_tags_dict = pos_tag_with_dict(text_to_tag)
print("PoS Tags with Dictionary:", pos_tags_dict)
```

```
PoS Tags with Dictionary: [('The', 'DT'), ('quick', 'JJ'), ('brown', 'JJ'), ('fox', 'NN'), ('jumps', 'NN'), ('over', 'IN'), ('the',
```

4. Train Unigram, Bi-gram, and Trigram Taggers

```
from nltk.corpus import brown
nltk.download('brown')
from nltk.tag import UnigramTagger, BigramTagger, TrigramTagger

# Training Data
tagged_sents = brown.tagged_sents(categories='news')

# Unigram Tagger
unigram_tagger = UnigramTagger(tagged_sents)

# Bigram Tagger
bigram_tagger = BigramTagger(tagged_sents)

# Trigram Tagger
trigram_tagger = TrigramTagger(tagged_sents)

# Testing
test_sentence = "The cat sat on the mat."
tokens = nltk.word_tokenize(test_sentence)
print("Unigram Tagger:", unigram_tagger.tag(tokens))
print("Bigram Tagger:", bigram_tagger.tag(tokens))
print("Trigram Tagger:", trigram_tagger.tag(tokens))
```

```
[nltk_data] Downloading package brown to /root/nltk_data...
[nltk_data]   Package brown is already up-to-date!
Unigram Tagger: [('The', 'AT'), ('cat', None), ('sat', 'VBD'), ('on', 'IN'), ('the', 'AT'), ('mat', 'NN'), ('.', '.')]
Bigram Tagger: [('The', 'AT'), ('cat', None), ('sat', None), ('on', None), ('the', None), ('mat', None), ('.', None)]
Trigram Tagger: [('The', 'AT'), ('cat', None), ('sat', None), ('on', None), ('the', None), ('mat', None), ('.', None)]
```

Name: Sanket S. Fulzele

Roll No: 371018

PRN: 22110728

Div: A

Comparison between Deep parsing and Shallow Parsing

Here's a brief discussion comparing deep parsing and shallow parsing in tabular form:

Aspect	Deep Parsing	Shallow Parsing
Scope	Analyzes the entire sentence or discourse	Focuses on smaller syntactic units or phrases
Granularity	Fine-grained analysis of sentence structure	Coarser analysis, often based on parts of speech tags or chunks
Information	Captures detailed syntactic relationships	Emphasizes on identifying phrases and entities
Complexity	More computationally intensive	Less computationally intensive
Output	Produces parse trees or dependency graphs	Outputs chunks or phrases
Applications	Machine translation, syntax-driven applications	Information extraction, named entity recognition

Implementation

```
import nltk
# nltk.download('maxent_ne_chunker')
# nltk.download('words')

# Customized Grammar
custom_grammar = r"""
  NP: {<DT>?<JJ>*<NN>}  # Noun Phrase
  AP: {<RB>?<JJ>+}      # Adjective Phrase
  AdvP: {<RB>+}         # Adverb Phrase
  VP: {<VB.*><NP|PP>*}  # Verb Phrase
"""
# Create a chunk parser with the customized grammar
chunk_parser = nltk.RegexpParser(custom_grammar)

# Sample sentence
sentence = [("The", "DT"), ("quick", "JJ"), ("brown", "JJ"), ("fox", "NN"), ("jumps", "VBZ"), ("over", "IN"),
            ("the", "DT"), ("lazy", "JJ"), ("dog", "NN")]

# Perform chunking using the custom grammar
chunks_custom = chunk_parser.parse(sentence)
print("Customized Chunking:", chunks_custom)

# Perform chunking using inbuilt functions

chunks_builtin = nltk.ne_chunk(sentence)
print("Built-in Chunking:", chunks_builtin)

# Compare outputs
print("\nCustomized Chunking:")
for subtree in chunks_custom.subtrees():
    if subtree.label() in ['NP', 'AP', 'AdvP', 'VP']:
        print(subtree.label(), ":", " ".join(word for word, pos in subtree.leaves()))

print("\nBuilt-in Chunking:")
print(chunks_builtin)
```

```
Customized Chunking: (S
  (NP The/DT quick/JJ brown/JJ fox/NN)
  (VP jumps/VBZ)
  over/IN
  (NP the/DT lazy/JJ dog/NN))
Built-in Chunking: (S
  The/DT
  quick/JJ
  brown/JJ
  fox/NN
  jumps/VBZ
  over/IN
  the/DT
  lazy/JJ
  dog/NN)

Customized Chunking:
NP : The quick brown fox
VP : jumps
NP : the lazy dog

Built-in Chunking:
(S
  The/DT
  quick/JJ
  brown/JJ
  fox/NN
  jumps/VBZ
  over/IN
  the/DT
  lazy/JJ
  dog/NN)
```

By comparing the outputs, you can analyze how well your customized grammar captures the desired chunks compared to NLTK's built-in chunking functions. Adjustments to the grammar and patterns may be needed based on the specific linguistic structures you want to capture.

```
Start coding or generate with AI.
```

Name: Sanket S. Fulzele

Roll No: 371018

PRN: 22110728

Div: A

The CKY (Cocke-Kasami-Younger) algorithm and its probabilistic variant, probabilistic CKY (PCYK), are parsing algorithms used in natural language processing (NLP) for syntactic parsing. Both algorithms are based on dynamic programming techniques and are particularly efficient for parsing context-free grammars (CFGs).

CKY Algorithm

The CKY algorithm is a bottom-up parsing algorithm that constructs a parse tree for a given sentence based on a CFG. It uses dynamic programming to efficiently compute and store partial parse results, reducing the time complexity of parsing.

Key Steps:

1. Initialization: Initialize a parse table with cells representing different spans of the sentence.
2. Parsing Table Filling: Fill in the table bottom-up by combining smaller spans to create larger spans based on CFG rules.
3. Backtracking: Trace back through the table to reconstruct the parse tree. The CKY algorithm is efficient and can handle ambiguous grammars and sentences with multiple parse trees. However, it requires the CFG to be in Chomsky Normal Form (CNF), where production rules are either of the form A -> BC or A -> w, where A, B, C are non-terminals and w is a terminal.

Probabilistic CKY (PCYK) Algorithm

The PCYK algorithm extends the CKY algorithm to handle probabilistic context-free grammars (PCFGs), where each production rule has associated probabilities. PCYK computes the most probable parse tree for a given sentence based on these probabilities.

Key Differences:

1. Probabilistic Scoring: PCYK assigns probabilities to each parse tree and selects the most probable parse tree based on these scores.
2. Parsing Table Initialization: The parse table in PCYK includes probabilities for each cell, representing the probability of generating a span with a specific non-terminal.
3. Probabilistic Parsing: During parsing table filling, PCYK computes probabilities for each possible split and combines probabilities according to CFG rules and production probabilities.

Comparison and Applications

1. Efficiency: Both CKY and PCYK are efficient parsing algorithms suitable for parsing CFGs and PCFGs, respectively.

2. Ambiguity Handling: CKY and PCYK can handle ambiguous grammars and sentences with multiple parse trees.
3. Probabilistic Modeling: PCYK is particularly useful for probabilistic parsing tasks, such as statistical parsing and machine translation, where probabilities play a crucial role in selecting the most likely parse tree.

```
import nltk

# Define CFG rules
cfg_rules = """
    S -> NP VP
    NP -> Det N | Det N PP | 'I'
    VP -> V NP | VP PP
    PP -> P NP
    Det -> 'a' | 'an' | 'the' | 'my'|'The'
    N -> 'dog' | 'cat' | 'ball' | 'park'|'bone'
    V -> 'chased' | 'ate' | 'threw'
    P -> 'in' | 'on' | 'at'
"""

# Create a CFG parser
cfg_parser = nltk.ChartParser(nltk.CFG.fromstring(cfg_rules))

# Define sentences to parse
sentences = [
    "I chased the dog in the park",
    "The cat ate my ball",
    "The dog chased the cat in the park",
    "I threw the ball at the cat",
    "The dog ate a bone"
]

# Define a function to parse sentences using CKY algorithm
def parse_sentence(sentence):
    tokens = nltk.word_tokenize(sentence)
    parsed_trees = cfg_parser.parse(tokens)
    return list(parsed_trees)

# Parse and print the parse trees for each sentence
for sentence in sentences:
    print(f"Sentence: {sentence}")
    parse_trees = (parse_sentence(sentence))
    for tree in parse_trees:
        print(tree)
    print()
```

```
Sentence: I chased the dog in the park
(S
  (NP I)
  (VP
    (VP (V chased) (NP (Det the) (N dog)))
    (PP (P in) (NP (Det the) (N park)))))
(S
  (NP I)
  (VP
    (V chased)
    (NP (Det the) (N dog) (PP (P in) (NP (Det the) (N park))))))

Sentence: The cat ate my ball
(S (NP (Det The) (N cat)) (VP (V ate) (NP (Det my) (N ball))))

Sentence: The dog chased the cat in the park
(S
  (NP (Det The) (N dog))
  (VP
    (VP (V chased) (NP (Det the) (N cat)))
```

```
    (PP (P in) (NP (Det the) (N park)))))
(S
  (NP (Det The) (N dog))
  (VP
    (V chased)
    (NP (Det the) (N cat) (PP (P in) (NP (Det the) (N park))))))

Sentence: I threw the ball at the cat
(S
  (NP I)
  (VP
    (VP (V threw) (NP (Det the) (N ball)))
    (PP (P at) (NP (Det the) (N cat)))))
(S
  (NP I)
  (VP
    (V threw)
    (NP (Det the) (N ball) (PP (P at) (NP (Det the) (N cat))))))

Sentence: The dog ate a bone
(S (NP (Det The) (N dog)) (VP (V ate) (NP (Det a) (N bone))))
```

Name: Sanket S. Fulzele

Roll No: 371018

PRN: 22110728

Div: A

Objective:
Perform single-word, multi-word based and polarity-based sentiment analysis.

1. Brief discussion on approaches to perform sentiment analysis.

Sentiment analysis, also known as opinion mining, is the process of identifying and extracting sentiment or opinions expressed in text data. There are several approaches to perform sentiment analysis, each with its advantages and limitations. Here's a brief discussion on some common approaches:

1. Lexicon-Based Approach:

- Lexicon-based sentiment analysis relies on predefined dictionaries or lexicons containing words annotated with sentiment scores (e.g., positive, negative, neutral).

- Each word in the text is matched against the lexicon, and the sentiment score is aggregated to determine the overall sentiment of the text.

- This approach is relatively simple and efficient but may struggle with context-dependent sentiments and newly coined words do not present in the lexicon.

2. Machine Learning Approach:

- Machine learning-based sentiment analysis involves training a classifier on labelled text data, where each sample is annotated with its corresponding sentiment label (e.g., positive, negative, neutral).

- Various machine learning algorithms such as Naive Bayes, Support Vector Machines (SVM), and deep learning models like Recurrent Neural Networks (RNNs) or Convolutional Neural Networks (CNNs) can be used for classification.

- This approach can capture complex patterns in the data and adapt to different domains but requires a large amount of labelled training data and may be computationally expensive.

3. Hybrid Approach:

- The hybrid approach combines both lexicon-based and machine learning-based methods to leverage their respective strengths.

- Lexicon-based methods can be used for feature engineering or as part of a preprocessing step, while machine learning models can be trained on the engineered features.

- This approach aims to achieve better performance by combining the advantages of both approaches.

4. Aspect-Based Sentiment Analysis:

- Aspect-based sentiment analysis goes beyond overall sentiment polarity and focuses on identifying sentiment towards specific aspects or features mentioned in the text.

- It involves extracting aspect terms (e.g., product features, service attributes) and analysing sentiment towards each aspect individually.

- This approach provides more granular insights into the opinions expressed in the text and is useful for tasks like product review analysis and social media monitoring.

5. Rule-Based Approach:

- Rule-based sentiment analysis relies on predefined rules or patterns to identify sentiment cues and infer sentiment from the text.

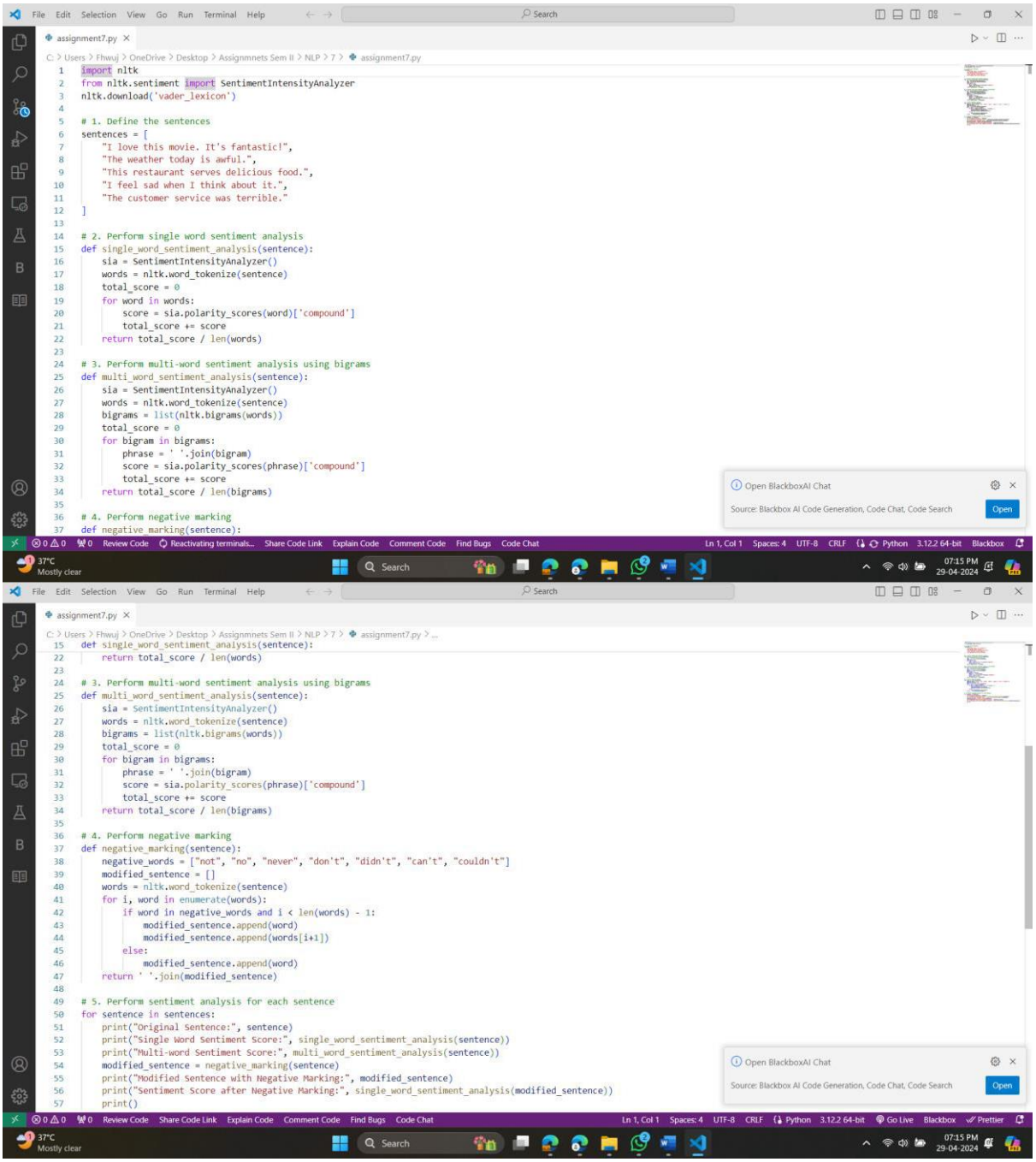
- Rules can be based on linguistic patterns, syntactic structures, or domain-specific knowledge.

- While rule-based approaches offer transparency and interpretability, they may be limited in capturing nuanced sentiments and require manual effort to design and maintain the rules.

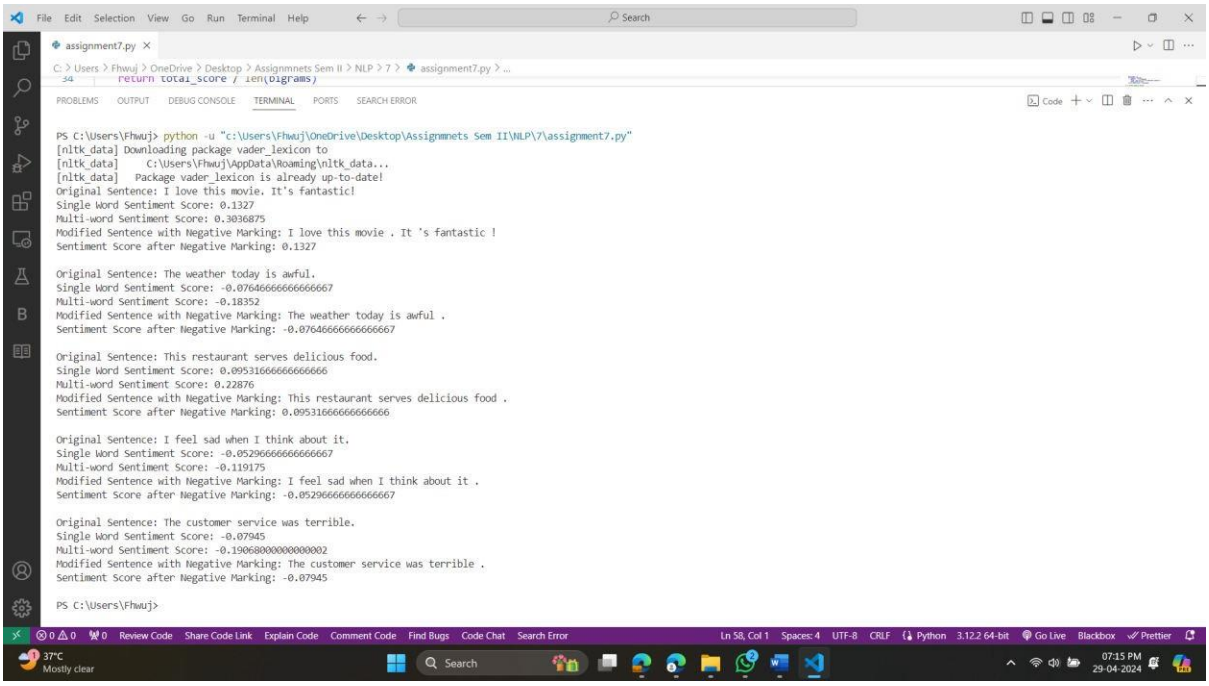
Overall, the choice of approach depends on factors such as the availability of labelled data, computational resources, and the specific requirements of the sentiment analysis task. It's common to

experiment with multiple approaches and techniques to achieve the best results for a given application.

2. Step wise code and output.



```python
import nltk
from nltk.sentiment import SentimentIntensityAnalyzer
nltk.download('vader_lexicon')

# 1. Define the sentences
sentences = [
    "I love this movie. It's fantastic!",
    "The weather today is awful.",
    "This restaurant serves delicious food.",
    "I feel sad when I think about it.",
    "The customer service was terrible."
]

# 2. Perform single word sentiment analysis
def single_word_sentiment_analysis(sentence):
    sia = SentimentIntensityAnalyzer()
    words = nltk.word_tokenize(sentence)
    total_score = 0
    for word in words:
        score = sia.polarity_scores(word)['compound']
        total_score += score
    return total_score / len(words)

# 3. Perform multi-word sentiment analysis using bigrams
def multi_word_sentiment_analysis(sentence):
    sia = SentimentIntensityAnalyzer()
    words = nltk.word_tokenize(sentence)
    bigrams = list(nltk.bigrams(words))
    total_score = 0
    for bigram in bigrams:
        phrase = ' '.join(bigram)
        score = sia.polarity_scores(phrase)['compound']
        total_score += score
    return total_score / len(bigrams)

# 4. Perform negative marking
def negative_marking(sentence):
    negative_words = ["not", "no", "never", "don't", "didn't", "can't", "couldn't"]
    modified_sentence = []
    words = nltk.word_tokenize(sentence)
    for i, word in enumerate(words):
        if word in negative_words and i < len(words) - 1:
            modified_sentence.append(word)
            modified_sentence.append(words[i+1])
        else:
            modified_sentence.append(word)
    return ' '.join(modified_sentence)

# 5. Perform sentiment analysis for each sentence
for sentence in sentences:
    print("Original Sentence:", sentence)
    print("Single Word Sentiment Score:", single_word_sentiment_analysis(sentence))
    print("Multi-word Sentiment Score:", multi_word_sentiment_analysis(sentence))
    modified_sentence = negative_marking(sentence)
    print("Modified Sentence with Negative Marking:", modified_sentence)
    print("Sentiment Score after Negative Marking:", single_word_sentiment_analysis(modified_sentence))
    print()
```


File Edit Selection View Go Run Terminal Help
assignment7.py
C: > Users > Fhwuj > OneDrive > Desktop > Assignmnets Sem II > NLP > 7 > assignment7.py > ...
return total_score / len(bigrams)
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SEARCH ERROR
PS C:\Users\Fhwuj> python -u "c:\Users\Fhwuj\OneDrive\Desktop\Assignmnets Sem II\NLP\7\assignment7.py"
[nltk_data] Downloading package vader_lexicon to
[nltk_data] C:\Users\Fhwuj\AppData\Roaming\nltk_data...
[nltk_data] Package vader_lexicon is already up-to-date!
Original Sentence: I love this movie. It's fantastic!
Single Word Sentiment Score: 0.1327
Multi-word Sentiment Score: 0.3036875
Modified Sentence with Negative Marking: I love this movie . It 's fantastic !
Sentiment Score after Negative Marking: 0.1327
Original Sentence: The weather today is awful.
Single Word Sentiment Score: -0.07646666666666667
Multi-word Sentiment Score: -0.18352
Modified Sentence with Negative Marking: The weather today is awful .
Sentiment Score after Negative Marking: -0.07646666666666667
Original Sentence: This restaurant serves delicious food.
Single Word Sentiment Score: 0.09531666666666666
Multi-word Sentiment Score: 0.22876
Modified Sentence with Negative Marking: This restaurant serves delicious food .
Sentiment Score after Negative Marking: 0.09531666666666666
Original Sentence: I feel sad when I think about it.
Single Word Sentiment Score: -0.05296666666666667
Multi-word Sentiment Score: -0.119175
Modified Sentence with Negative Marking: I feel sad when I think about it .
Sentiment Score after Negative Marking: -0.05296666666666667
Original Sentence: The customer service was terrible.
Single Word Sentiment Score: -0.07945
Multi-word Sentiment Score: -0.19068000000000002
Modified Sentence with Negative Marking: The customer service was terrible .
Sentiment Score after Negative Marking: -0.07945
PS C:\Users\Fhwuj>
Review Code Share Code Link Explain Code Comment Code Find Bugs Code Chat Search Error
Ln 58, Col 1 Spaces: 4 UTF-8 CRLF Python 3.12.2 64-bit Go Live Blackbox Prettier
37°C
Mostly clear
Search
07:15 PM
29-04-2024

Name: Sanket S. Fulzele

Roll No: 371018

PRN: 22110728

Div: A

Objective:
Develop a language model to predict next best word.

1. Brief discussion on language modelling

Language modeling is a fundamental task in natural language processing (NLP) that involves predicting the probability distribution of words or sequences of words in a given text. It plays a crucial role in various NLP applications such as machine translation, speech recognition, and text generation. Here's a brief discussion on language modeling:

1. **Definition**:

- Language modeling refers to the process of learning the structure and patterns of natural language text to predict the likelihood of observing a sequence of words.

- The main goal of language modeling is to capture the underlying structure of a language and model the relationships between words in a corpus.

2. **Approaches**:

- Statistical Language Models: Statistical language models estimate the probability of word sequences based on statistical properties of the training corpus. They include n-gram models, where the probability of a word is conditioned on the previous n-1 words.

- Neural Language Models: Neural language models leverage deep learning techniques, such as recurrent neural networks (RNNs), long short-term memory (LSTM) networks, and transformers, to learn distributed representations of words and capture complex language patterns.

3. **Applications**:

- Text Generation: Language models can generate coherent and contextually relevant text by predicting the next word or sequence of words given a context.

- Speech Recognition: Language models help improve the accuracy of speech recognition systems by providing contextually relevant word predictions.

- Machine Translation: Language models aid in generating fluent and natural translations by modeling the target language's syntax and semantics.

- Information Retrieval: Language models can be used to rank documents or passages based on their relevance to a given query by estimating the probability of observing the query words in the documents.

4. **Evaluation**:

- Perplexity: Perplexity is a common metric used to evaluate the performance of language models. It measures how well the model predicts a given sequence of words and quantifies the uncertainty of the model's predictions.

- Human Evaluation: Human judgment can also be used to evaluate the quality of generated text or translations produced by language models, assessing factors such as fluency, coherence, and relevance.

5. **Challenges**:

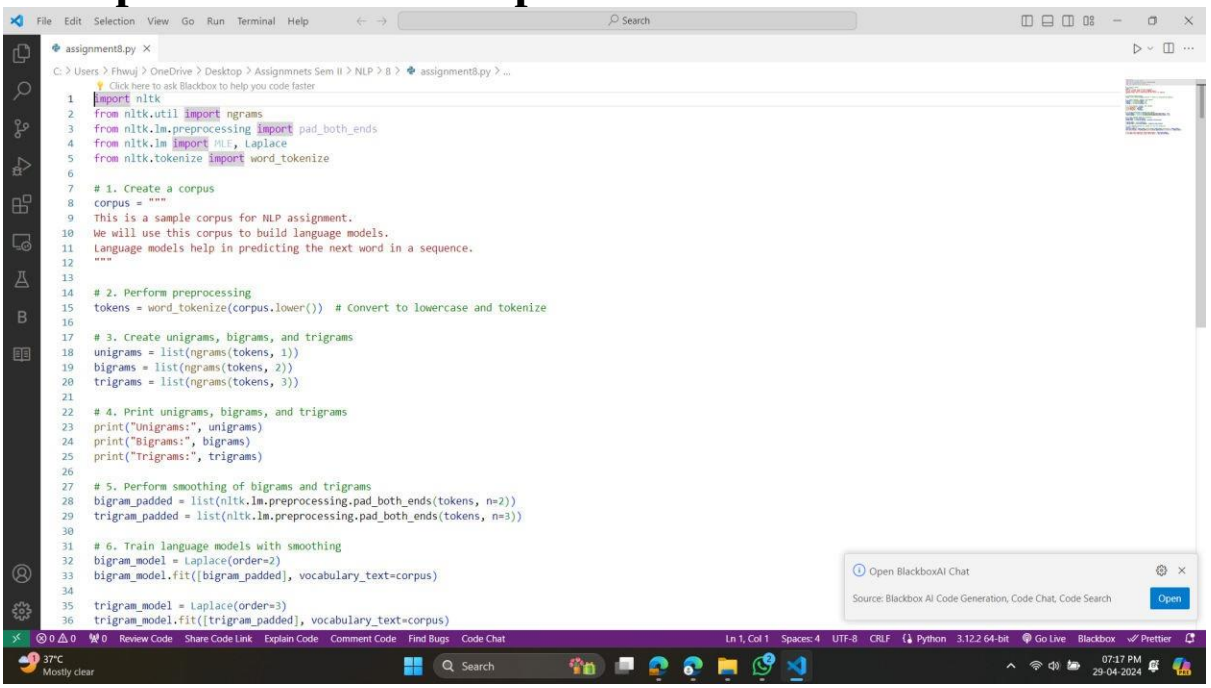
- Data Sparsity: Language models may struggle with rare or unseen words and phrases, leading to poor generalization on out-of-vocabulary terms.

- Long-Term Dependencies: Traditional n-gram models have difficulty capturing long-range dependencies and contextual information beyond a fixed window size.

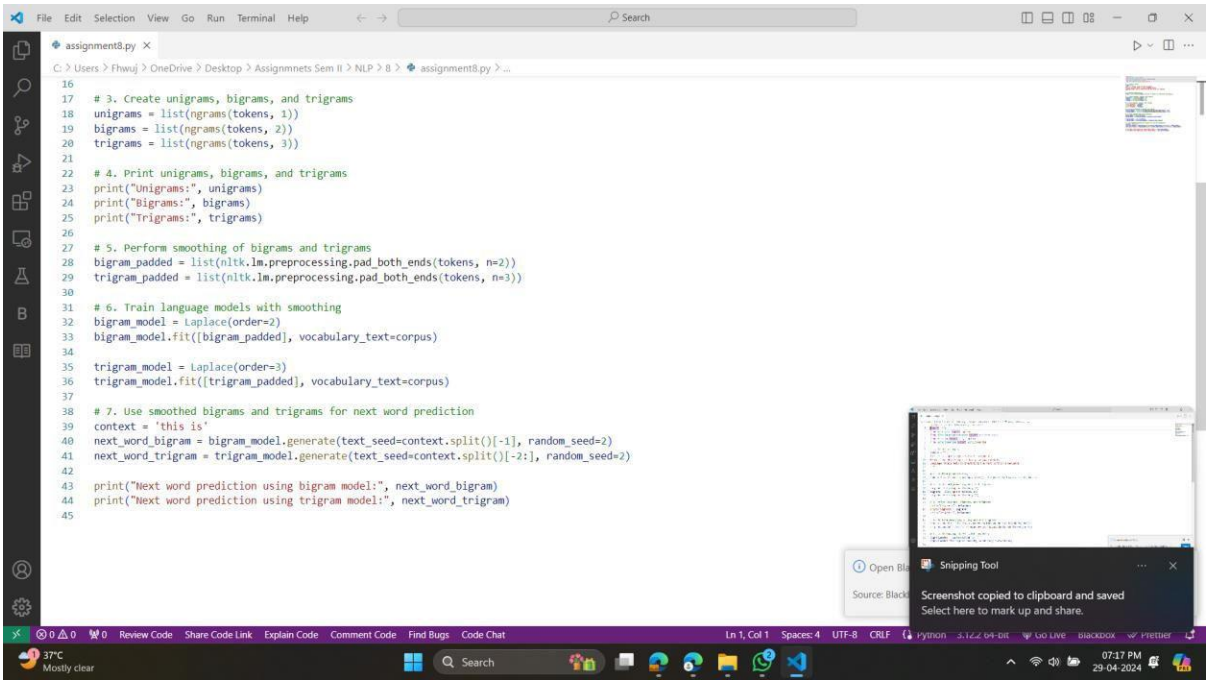
- Overfitting: Neural language models can suffer from overfitting, especially when trained on small datasets or when the model architecture is too complex.

Overall, language modeling is a foundational task in NLP, with numerous applications and ongoing research to address its challenges and improve its performance across various domains.

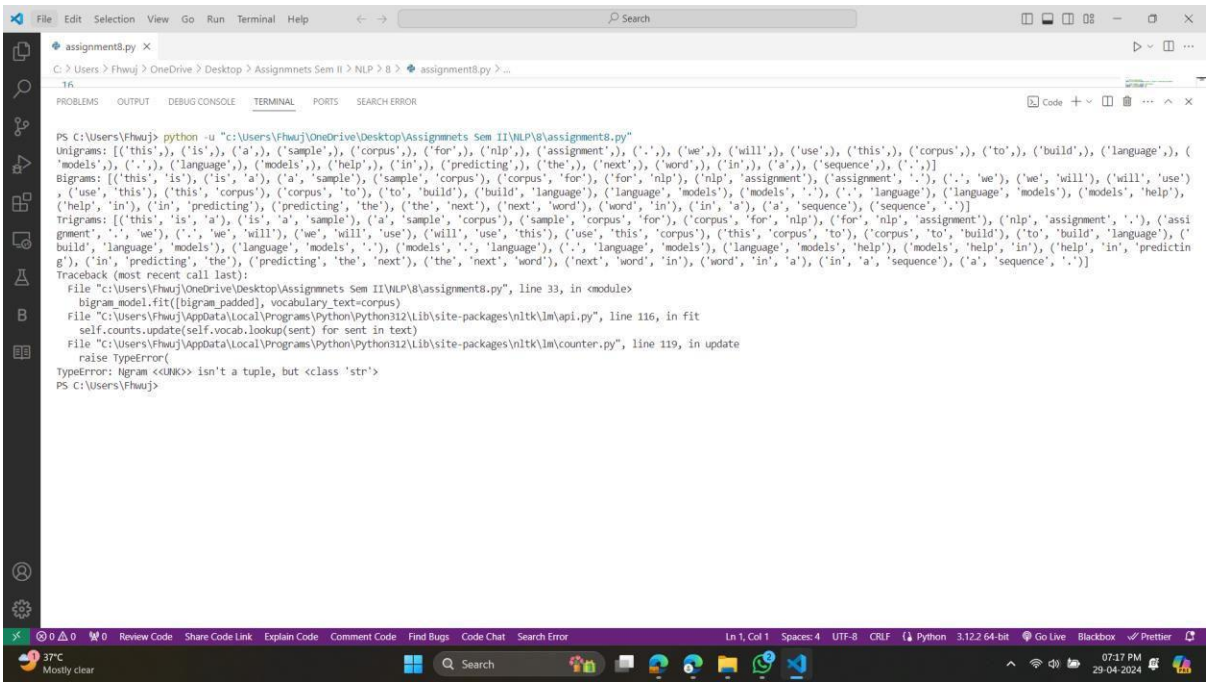
2. Step wise code and output.



```python
import nltk
from nltk.util import ngrams
from nltk.lm.preprocessing import pad_both_ends
from nltk.lm import MLE, Laplace
from nltk.tokenize import word_tokenize

# 1. Create a corpus
corpus = """
This is a sample corpus for NLP assignment.
We will use this corpus to build language models.
Language models help in predicting the next word in a sequence.
"""

# 2. Perform preprocessing
tokens = word_tokenize(corpus.lower())  # Convert to lowercase and tokenize

# 3. Create unigrams, bigrams, and trigrams
unigrams = list(ngrams(tokens, 1))
bigrams = list(ngrams(tokens, 2))
trigrams = list(ngrams(tokens, 3))

# 4. Print unigrams, bigrams, and trigrams
print("Unigrams:", unigrams)
print("Bigrams:", bigrams)
print("Trigrams:", trigrams)

# 5. Perform smoothing of bigrams and trigrams
bigram_padded = list(nltk.lm.preprocessing.pad_both_ends(tokens, n=2))
trigram_padded = list(nltk.lm.preprocessing.pad_both_ends(tokens, n=3))

# 6. Train language models with smoothing
bigram_model = Laplace(order=2)
bigram_model.fit([bigram_padded], vocabulary_text=corpus)

trigram_model = Laplace(order=3)
trigram_model.fit([trigram_padded], vocabulary_text=corpus)
```



```python
# 3. Create unigrams, bigrams, and trigrams
unigrams = list(ngrams(tokens, 1))
bigrams = list(ngrams(tokens, 2))
trigrams = list(ngrams(tokens, 3))

# 4. Print unigrams, bigrams, and trigrams
print("Unigrams:", unigrams)
print("Bigrams:", bigrams)
print("Trigrams:", trigrams)

# 5. Perform smoothing of bigrams and trigrams
bigram_padded = list(nltk.lm.preprocessing.pad_both_ends(tokens, n=2))
trigram_padded = list(nltk.lm.preprocessing.pad_both_ends(tokens, n=3))

# 6. Train language models with smoothing
bigram_model = Laplace(order=2)
bigram_model.fit([bigram_padded], vocabulary_text=corpus)

trigram_model = Laplace(order=3)
trigram_model.fit([trigram_padded], vocabulary_text=corpus)

# 7. Use smoothed bigrams and trigrams for next word prediction
context = 'this is'
next_word_bigram = bigram_model.generate(text_seed=context.split()[-1], random_seed=2)
next_word_trigram = trigram_model.generate(text_seed=context.split()[-2:], random_seed=2)

print("Next word prediction using bigram model:", next_word_bigram)
print("Next word prediction using trigram model:", next_word_trigram)
```


File Edit Selection View Go Run Terminal Help
Search
assignment8.py
C: > Users > Fhwuj > OneDrive > Desktop > Assignmnets Sem II > NLP > 8 > assignment8.py > ...
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SEARCH ERROR
Code
PS C:\Users\Fhwuj> python -u "c:\Users\Fhwuj\OneDrive\Desktop\Assignmnets Sem II\NLP\8\assignment8.py"
Unigrams: [('this',), ('is',), ('a',), ('sample',), ('corpus',), ('for',), ('nlp',), ('assignment',), ('.',), ('we',), ('will',), ('use',), ('this',), ('corpus',), ('to',), ('build',), ('language',), ('models',), ('.',), ('language',), ('models',), ('help',), ('in',), ('predicting',), ('the',), ('next',), ('word',), ('in',), ('a',), ('sequence',), ('.',)]
Bigrams: [('this', 'is'), ('is', 'a'), ('a', 'sample'), ('sample', 'corpus'), ('corpus', 'for'), ('for', 'nlp'), ('nlp', 'assignment'), ('assignment', '.'), ('.', 'we'), ('we', 'will'), ('will', 'use'), ('use', 'this'), ('this', 'corpus'), ('corpus', 'to'), ('to', 'build'), ('build', 'language'), ('language', 'models'), ('models', '.'), ('.', 'language'), ('language', 'models'), ('models', 'help'), ('help', 'in'), ('in', 'predicting'), ('predicting', 'the'), ('the', 'next'), ('next', 'word'), ('word', 'in'), ('in', 'a'), ('a', 'sequence'), ('sequence', '.')]
Trigrams: [('this', 'is', 'a'), ('is', 'a', 'sample'), ('a', 'sample', 'corpus'), ('sample', 'corpus', 'for'), ('corpus', 'for', 'nlp'), ('for', 'nlp', 'assignment'), ('nlp', 'assignment', '.'), ('assignment', '.', 'we'), ('.', 'we', 'will'), ('we', 'will', 'use'), ('will', 'use', 'this'), ('use', 'this', 'corpus'), ('this', 'corpus', 'to'), ('corpus', 'to', 'build'), ('to', 'build', 'language'), ('build', 'language', 'models'), ('language', 'models', '.'), ('models', '.', 'language'), ('.', 'language', 'models'), ('language', 'models', 'help'), ('models', 'help', 'in'), ('help', 'in', 'predicting'), ('in', 'predicting', 'the'), ('predicting', 'the', 'next'), ('the', 'next', 'word'), ('next', 'word', 'in'), ('word', 'in', 'a'), ('in', 'a', 'sequence'), ('a', 'sequence', '.')]
Traceback (most recent call last):
File "c:\Users\Fhwuj\OneDrive\Desktop\Assignmnets Sem II\NLP\8\assignment8.py", line 33, in <module>
bigram_model.fit([bigram_padded], vocabulary_text=corpus)
File "C:\Users\Fhwuj\AppData\Local\Programs\Python\Python312\Lib\site-packages\nltk\lm\api.py", line 116, in fit
self.counts.update(self.vocab.lookup(sent) for sent in text)
File "C:\Users\Fhwuj\AppData\Local\Programs\Python\Python312\Lib\site-packages\nltk\lm\counter.py", line 119, in update
raise TypeError(
TypeError: Ngram <<UNK>> isn't a tuple, but <class 'str'>
PS C:\Users\Fhwuj>
Review Code Share Code Link Explain Code Comment Code Find Bugs Code Chat Search Error
Ln 1, Col 1 Spaces: 4 UTF-8 CRLF Python 3.12.2 64-bit Go Live Blackbox Prettier
37°C
Mostly clear
Search
07:17 PM
29-04-2024

Name: Sanket S. Fulzele

Roll No: 371018

PRN: 22110728

Div: A

Objective:

Implement CKY algorithm for deep parsing.

1. Brief discussion Name Entities and their types

Name Entities and their Types
Name Entities (NE) are pieces of text that represent real-world objects or concepts. These can be people, organizations, locations,dates, monetary values, percentages, etc. NER is the process of automatically identifying and classifying these entities within a text.

Here's a breakdown of Name Entities and their types:

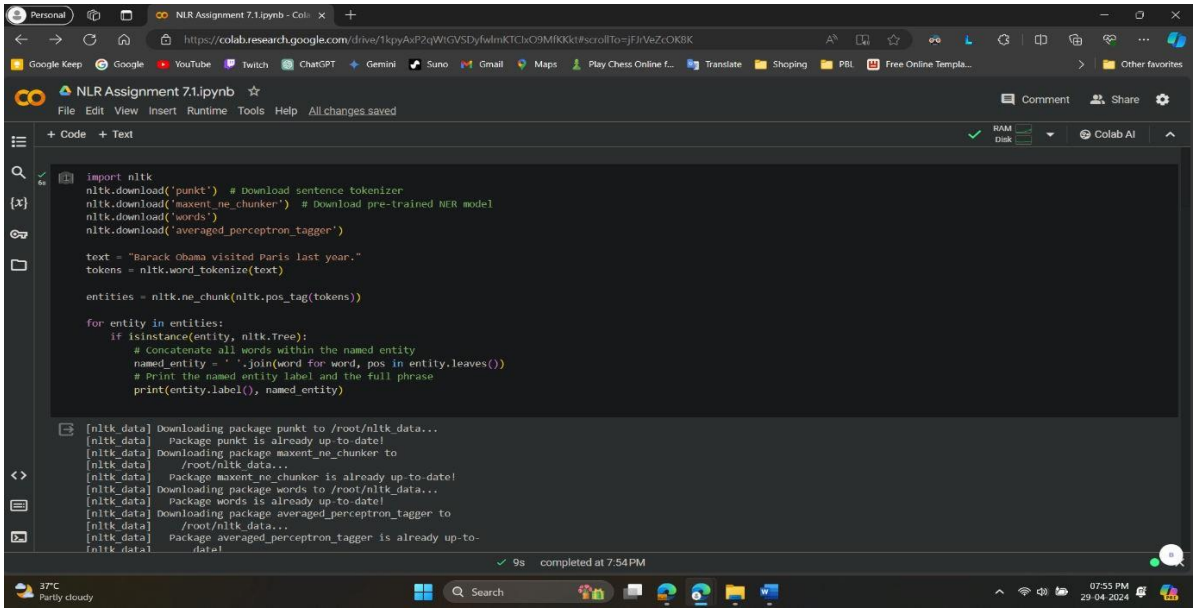
- Types of Name Entities:
    - People: Names of individuals (e.g., Barack Obama, Albert Einstein).
    - Organizations: Names of companies, government agencies, NGOs, etc. (e.g., Apple, World Health Organization).
    - Locations: Geographical entities like countries, cities, landmarks (e.g., France, Paris, Eiffel Tower).
    - Dates and Times: Specific points in time (e.g., 2023-10-26, 3:00 PM).
    - Monetary Values: Amounts of currency (e.g., $100, €25).
    - Percentages: Values represented as a percentage (e.g., 15%, 75%).
    - Other Types: Depending on the application, NER systems can be trained to recognize additional entity types like creative works (books, movies), vehicles, or even custom entities specific to a domain.
- Importance of NER:
    - Information Extraction: NER helps extract structured information from text for tasks like populating databases or knowledge graphs.
    - Machine Translation: NER helps identify entities that need to be translated consistently across languages.
    - Question Answering: NER helps identify entities relevant to answer user questions based on text.

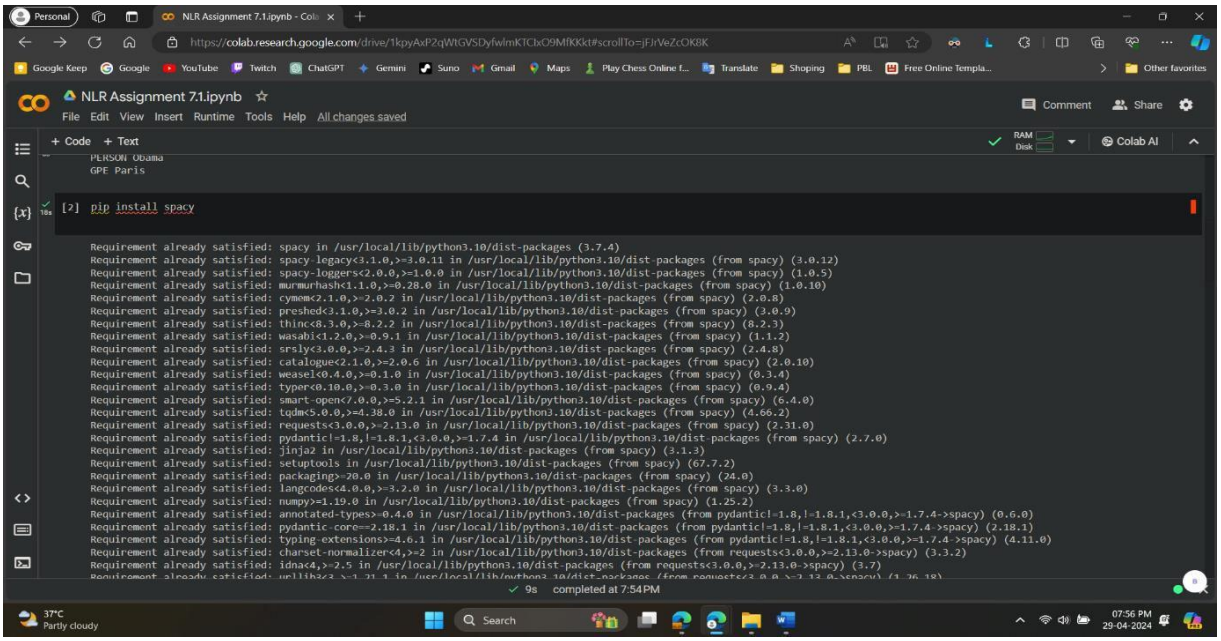
- Text Summarization: NER can help identify key entities to include in a concise summary.

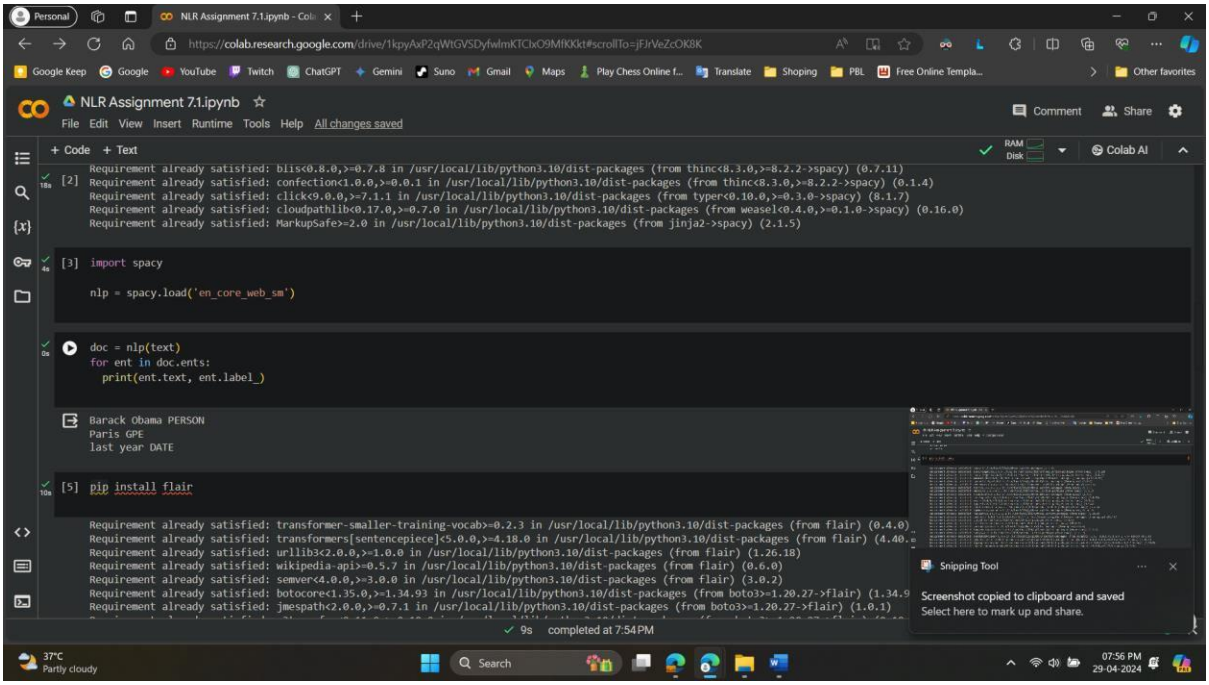
By recognizing and classifying named entities, we can unlock valuable information hidden within text data.

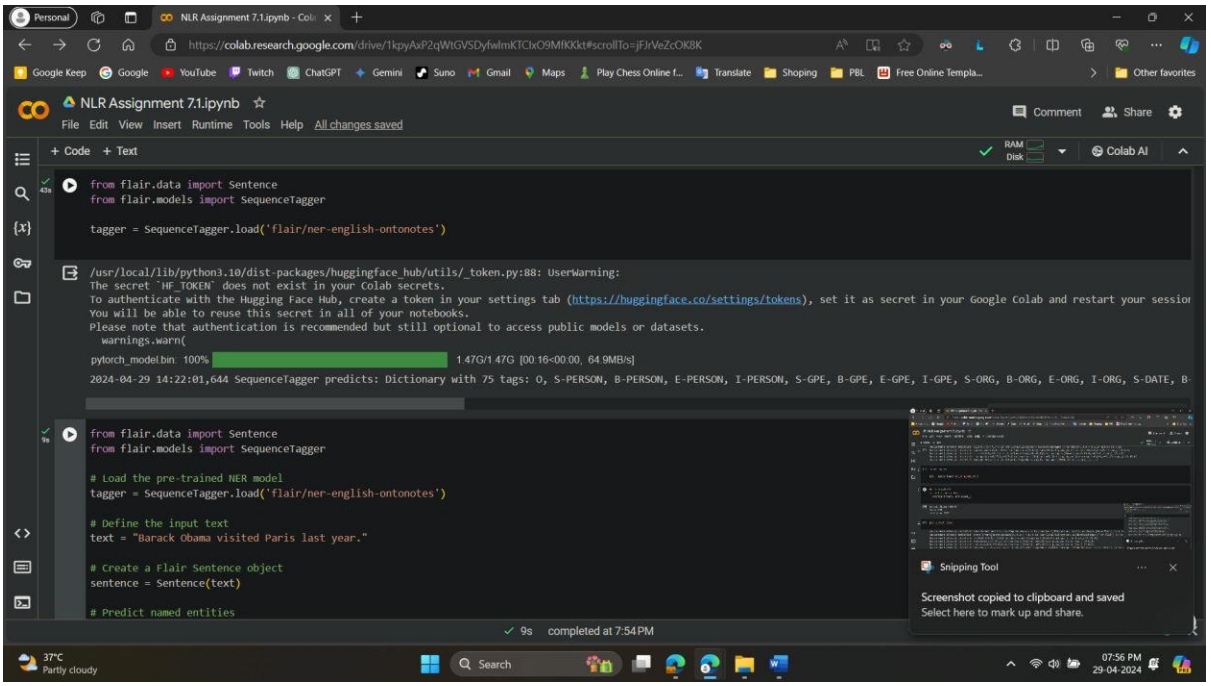
2. Step wise code and output.

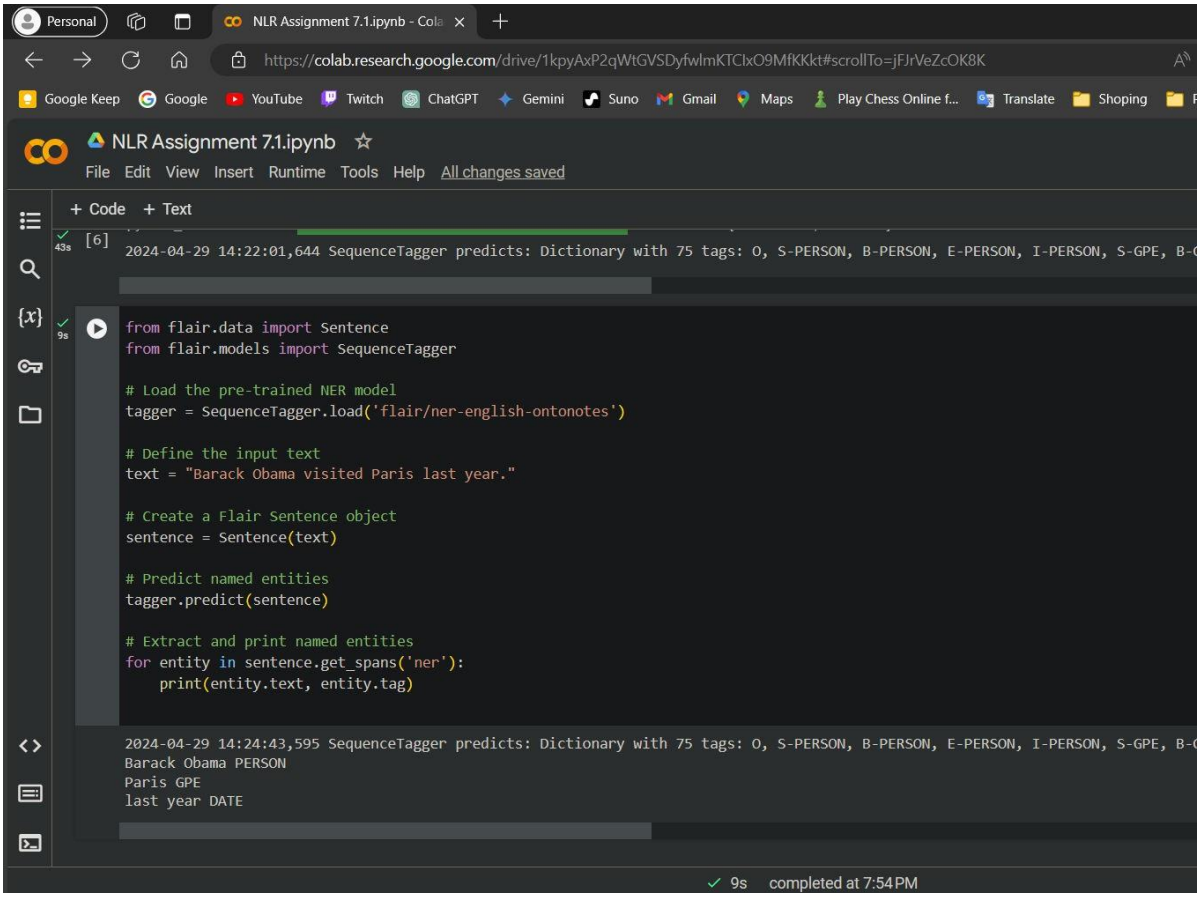
https://colab.research.google.com/drive/1kpyAxP2qWtGVSDyfwlmKTClxO9MfKKkt?usp=sharing




NLR Assignment 7.1.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
PERSON Obama
GPE Paris
[2] pip install spacy
Requirement already satisfied: spacy in /usr/local/lib/python3.10/dist-packages (3.7.4)
Requirement already satisfied: spacy-legacy<3.1.0,>=3.0.11 in /usr/local/lib/python3.10/dist-packages (from spacy) (3.0.12)
Requirement already satisfied: spacy-loggers<2.0.0,>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from spacy) (1.0.5)
Requirement already satisfied: murmurhash<1.1.0,>=0.28.0 in /usr/local/lib/python3.10/dist-packages (from spacy) (1.0.10)
Requirement already satisfied: cymem<2.1.0,>=2.0.2 in /usr/local/lib/python3.10/dist-packages (from spacy) (2.0.8)
Requirement already satisfied: preshed<3.1.0,>=3.0.2 in /usr/local/lib/python3.10/dist-packages (from spacy) (3.0.9)
Requirement already satisfied: thinc<8.3.0,>=8.2.2 in /usr/local/lib/python3.10/dist-packages (from spacy) (8.2.3)
Requirement already satisfied: wasabi<1.2.0,>=0.9.1 in /usr/local/lib/python3.10/dist-packages (from spacy) (1.1.2)
Requirement already satisfied: srsly<3.0.0,>=2.4.3 in /usr/local/lib/python3.10/dist-packages (from spacy) (2.4.8)
Requirement already satisfied: catalogue<2.1.0,>=2.0.6 in /usr/local/lib/python3.10/dist-packages (from spacy) (2.0.10)
Requirement already satisfied: weasel<0.4.0,>=0.1.0 in /usr/local/lib/python3.10/dist-packages (from spacy) (0.3.4)
Requirement already satisfied: typer<0.10.0,>=0.3.0 in /usr/local/lib/python3.10/dist-packages (from spacy) (0.9.4)
Requirement already satisfied: smart-open<7.0.0,>=5.2.1 in /usr/local/lib/python3.10/dist-packages (from spacy) (6.4.0)
Requirement already satisfied: tqdm<5.0.0,>=4.38.0 in /usr/local/lib/python3.10/dist-packages (from spacy) (4.66.2)
Requirement already satisfied: requests<3.0.0,>=2.13.0 in /usr/local/lib/python3.10/dist-packages (from spacy) (2.31.0)
Requirement already satisfied: pydantic!=1.8,!=1.8.1,<3.0.0,>=1.7.4 in /usr/local/lib/python3.10/dist-packages (from spacy) (2.7.0)
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-packages (from spacy) (3.1.3)
Requirement already satisfied: setuptools in /usr/local/lib/python3.10/dist-packages (from spacy) (67.7.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10/dist-packages (from spacy) (24.0)
Requirement already satisfied: langcodes<4.0.0,>=3.2.0 in /usr/local/lib/python3.10/dist-packages (from spacy) (3.3.0)
Requirement already satisfied: numpy>=1.19.0 in /usr/local/lib/python3.10/dist-packages (from spacy) (1.25.2)
Requirement already satisfied: annotated-types>=0.4.0 in /usr/local/lib/python3.10/dist-packages (from pydantic!=1.8,!=1.8.1,<3.0.0,>=1.7.4->spacy) (0.6.0)
Requirement already satisfied: pydantic-core==2.18.1 in /usr/local/lib/python3.10/dist-packages (from pydantic!=1.8,!=1.8.1,<3.0.0,>=1.7.4->spacy) (2.18.1)
Requirement already satisfied: typing-extensions>=4.6.1 in /usr/local/lib/python3.10/dist-packages (from pydantic!=1.8,!=1.8.1,<3.0.0,>=1.7.4->spacy) (4.11.0)
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/python3.10/dist-packages (from requests<3.0.0,>=2.13.0->spacy) (3.3.2)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/dist-packages (from requests<3.0.0,>=2.13.0->spacy) (3.7)
9s completed at 7:54PM
37°C Partly cloudy
Search
07:56 PM 29-04-2024


NLR Assignment 7.1.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
Requirement already satisfied: blis<0.8.0,>=0.7.8 in /usr/local/lib/python3.10/dist-packages (from thinc<8.3.0,>=8.2.2->spacy) (0.7.11)
[2] Requirement already satisfied: confection<1.0.0,>=0.0.1 in /usr/local/lib/python3.10/dist-packages (from thinc<8.3.0,>=8.2.2->spacy) (0.1.4)
Requirement already satisfied: click<9.0.0,>=7.1.1 in /usr/local/lib/python3.10/dist-packages (from typer<0.10.0,>=0.3.0->spacy) (8.1.7)
Requirement already satisfied: cloudpathlib<0.17.0,>=0.7.0 in /usr/local/lib/python3.10/dist-packages (from weasel<0.4.0,>=0.1.0->spacy) (0.16.0)
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10/dist-packages (from jinja2->spacy) (2.1.5)
[3] import spacy
nlp = spacy.load('en_core_web_sm')
doc = nlp(text)
for ent in doc.ents:
print(ent.text, ent.label_)
Barack Obama PERSON
Paris GPE
last year DATE
[5] pip install flair
Requirement already satisfied: transformer-smaller-training-vocab>=0.2.3 in /usr/local/lib/python3.10/dist-packages (from flair) (0.4.0)
Requirement already satisfied: transformers[sentencepiece]<5.0.0,>=4.18.0 in /usr/local/lib/python3.10/dist-packages (from flair) (4.40.
Requirement already satisfied: urllib3<2.0.0,>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from flair) (1.26.18)
Requirement already satisfied: wikipedia-api>=0.5.7 in /usr/local/lib/python3.10/dist-packages (from flair) (0.6.0)
Requirement already satisfied: semver<4.0.0,>=3.0.0 in /usr/local/lib/python3.10/dist-packages (from flair) (3.0.2)
Requirement already satisfied: botocore<1.35.0,>=1.34.93 in /usr/local/lib/python3.10/dist-packages (from boto3>=1.20.27->flair) (1.34.9
Requirement already satisfied: jmespath<2.0.0,>=0.7.1 in /usr/local/lib/python3.10/dist-packages (from boto3>=1.20.27->flair) (1.0.1)
Snipping Tool
Screenshot copied to clipboard and saved
Select here to mark up and share.
9s completed at 7:54PM
37°C Partly cloudy
Search
07:56 PM 29-04-2024


NLR Assignment 7.1.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
from flair.data import Sentence
from flair.models import SequenceTagger
tagger = SequenceTagger.load('flair/ner-english-ontonotes')
/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_token.py:88: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Google Colab and restart your session
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
warnings.warn(
pytorch_model.bin: 100% 1.47G/1.47G [00:16<00:00, 64.9MB/s]
2024-04-29 14:22:01,644 SequenceTagger predicts: Dictionary with 75 tags: O, S-PERSON, B-PERSON, E-PERSON, I-PERSON, S-GPE, B-GPE, E-GPE, I-GPE, S-ORG, B-ORG, E-ORG, I-ORG, S-DATE, B-
from flair.data import Sentence
from flair.models import SequenceTagger
# Load the pre-trained NER model
tagger = SequenceTagger.load('flair/ner-english-ontonotes')
# Define the input text
text = "Barack Obama visited Paris last year."
# Create a Flair Sentence object
sentence = Sentence(text)
# Predict named entities
Snipping Tool
Screenshot copied to clipboard and saved
Select here to mark up and share.
9s completed at 7:54 PM
37°C Partly cloudy
Search
07:56 PM 29-04-2024


NLR Assignment 7.1.ipynb - Cola
https://colab.research.google.com/drive/1kpyAxP2qWtGVSDyfwlmKTClxO9MfKKkt#scrollTo=jFJrVeZcOK8K
Google Keep Google YouTube Twitch ChatGPT Gemini Suno Gmail Maps Play Chess Online f... Translate Shoping
NLR Assignment 7.1.ipynb
File Edit View Insert Runtime Tools Help All changes saved
+ Code + Text
[6]
2024-04-29 14:22:01,644 SequenceTagger predicts: Dictionary with 75 tags: O, S-PERSON, B-PERSON, E-PERSON, I-PERSON, S-GPE, B-
from flair.data import Sentence
from flair.models import SequenceTagger
# Load the pre-trained NER model
tagger = SequenceTagger.load('flair/ner-english-ontonotes')
# Define the input text
text = "Barack Obama visited Paris last year."
# Create a Flair Sentence object
sentence = Sentence(text)
# Predict named entities
tagger.predict(sentence)
# Extract and print named entities
for entity in sentence.get_spans('ner'):
print(entity.text, entity.tag)
2024-04-29 14:24:43,595 SequenceTagger predicts: Dictionary with 75 tags: O, S-PERSON, B-PERSON, E-PERSON, I-PERSON, S-GPE, B-
Barack Obama PERSON
Paris GPE
last year DATE
9s completed at 7:54 PM