UNIVERSITY OF MUMBAI

DEPARTMENT OF COMPUTER SCIENCE

मुंबई विद्यापीठ
University of Mumbai
Re-accredited with A++ Grade
(CGPA 3.65) by NAAC (3rd Cycle 2021)

M.Sc. Computer Science - Semester II

(NEP 2020)

Natural Language Processing

JOURNAL

2023-2024

Seat No. _____

मुंबई विद्यापीठ
University of Mumbai
Re-accredited with A++ Grade
(CGPA 3.65) by NAAC (3rd Cycle 2021)

UNIVERSITY OF MUMBAI

DEPARTMENT OF COMPUTER SCIENCE

CERTIFICATE

This is to certify that the work entered in this journal was done in the University Department of Computer Sciencelaboratory by Mr./Ms._____Seat No. _____ for the course of M.Sc. (Computer Science) - Semester II (NEP 2020) during the academic year 2023- 2024 in a satisfactory manner.

_____ _____

Subject In-charge **Head of Department**

_____

External Examiner

INDEX

Practical-1: Write a program to implement sentence segmentation and word tokenization

Segmentation and word tokenization

In Natural Language Processing (NLP), segmentation and word tokenization are fundamental tasks that play a crucial role in preparing text data for analysis and processing. Here's a detailed overview of both concepts:

Segmentation refers to dividing a text into meaningful units, which can be sentences, words, or phrases, depending on the application. It is often the first step in text preprocessing. The two primary types of segmentation are:

Sentence Segmentation: Also known as sentence boundary detection or sentence splitting, it involves identifying the boundaries between sentences in a given text. This is typically done by looking for punctuation marks such as periods, exclamation marks, and question marks, but can also involve more complex rules and machine learning models to handle exceptions and ambiguities.

Input: "Hello world! How are you?" Output:

["Hello world!", "How are you?"]

Word Segmentation: Also known as word tokenization, it involves dividing a sentence into individual wordsor tokens. This is straightforward in languages where words are typically separated by spaces (e.g., English),but can be more challenging in languages without explicit word boundaries, such as Chinese or Thai.

Input: "Hello world!"

Output: ["Hello", "world", "!"]

Word tokenization is the process of splitting text into individual words or tokens. This step is crucial for many NLP tasks, including text classification, sentiment analysis, and machine translation. Tokenization strategiescan vary based on the language and the specific requirements of the task.

Types of Tokenization

Rule-based Tokenization: This approach uses predefined rules and regular expressions to split text into tokens.It is simple and fast but may not handle all linguistic nuances.

Regular expression to split on whitespace and punctuation.

Statistical Tokenization: This method uses probabilistic models, such as Hidden Markov Models (HMMs), to determine token boundaries based on the likelihood of sequences of characters forming valid tokens.

Dictionary-based Tokenization: This technique involves looking up substrings in a dictionary of known words. It is commonly used in languages like Chinese, where words are not separated by spaces. **Subword Tokenization:** Approaches like Byte Pair Encoding (BPE) and WordPiece involve breaking down words into smaller units, such as subwords or even characters. This is particularly useful for handling rare words, misspellings, and morphological variations.

Input: "unhappiness"

Output: ["un", "happiness"] (BPE)

Challenges in Tokenization

Ambiguity: Words can have different meanings based on context, and some languages have words that are notseparated by spaces.

Compound Words: Some languages use compound words that can be difficult to tokenize correctly.

Contractions and Hyphenation: Handling contractions (e.g., "don't") and hyphenated words (e.g., "mother-in-law") can be tricky.

Punctuation: Deciding whether punctuation should be treated as separate tokens or attached to words.

Tools and Libraries for Tokenization

Several NLP libraries provide tools for segmentation and tokenization:

NLTK (Natural Language Toolkit): A comprehensive library for working with human language data in Python.

Example: nltk.tokenize.sent_tokenize, nltk.tokenize.word_tokenize

spaCy: An open-source library for advanced NLP in Python, known for its high performance.

Example: spacy.tokens.Doc.sents, spacy.tokens.Doc

Stanford NLP: A suite of NLP tools developed by Stanford University, supporting many languages.

Example: StanfordTokenizer

Hugging Face Transformers: Provides tokenization utilities that are particularly useful for working withtransformer models.

Example: transformers.AutoTokenizer

Segmentation and word tokenization are essential preprocessing steps in NLP, enabling more advanced analysisand model building. The choice of segmentation and tokenization method depends on the specific requirements of the task, the characteristics of the language, and the desired balance between simplicity and accuracy.

CODE:

```
#pip intall nltk
#py -m pip install --upgrade pip
#nltk.download('punkt') #nltk.download('wordnet')

from nltk.tokenize import word_tokenize
text="God is Great! I won a lottery." print("The
words are",word_tokenize(text))

from nltk.tokenize import sent_tokenize text="God
is Great! I won a lottery." print("The sentences
are",sent_tokenize(text))
```

OUTPUT:

```
The words are ['God', 'is', 'Great', '!', 'I', 'won', 'a', 'lottery', '.']
The sentences are ['God is Great!', 'I won a lottery.']
```

Practical-2: Write a program to Implement stemming and lemmatization

Stemming and Lemmatization

Stemming and lemmatization are techniques used in Natural Language Processing (NLP) for text normalization.Both are used to reduce words to their base or root form, but they do so in different ways and with different goals. Here's a detailed explanation of both concepts:

Stemming is the process of reducing a word to its base or root form, typically by removing suffixes or prefixes.The resulting stem word may not be a real word but is intended to be a common base form.

Characteristics of Stemming

Algorithmic: Stemming uses simple, rule-based algorithms to chop off affixes (prefixes and suffixes). Itdoesn't take into account the context or meaning of the word.

Speed: Stemming algorithms are generally faster and less computationally intensive. **Accuracy**:

Stemming can be less accurate and may produce stems that are not actual words.**Common**

Stemming Algorithms

Porter Stemmer: One of the most widely used stemming algorithms, known for its simplicity and effectivenessin English.

Snowball Stemmer: An improved version of the Porter Stemmer, also known as the Porter2 stemmer. **Lancaster**

Stemmer: Another stemming algorithm, which is more aggressive and can produce shorter stems.Input: "running",

"runner", "ran"

Porter Stemmer Output: "run", "runner", "ran"

Snowball Stemmer Output: "run", "runner", "ran"

Lancaster Stemmer Output: "run", "run", "ran"

Lemmatization is the process of reducing a word to its base or dictionary form, known as the lemma. It takes into account the context and the morphological analysis of the word to ensure that the root form is a valid word in the language.

Characteristics of Lemmatization

Contextual: Lemmatization considers the context and part of speech of the word to accurately convert it to itsbase form.

Accuracy: Generally more accurate than stemming, as it ensures that the lemma is a real word.

Complexity: More computationally intensive than stemming, as it requires access to a dictionary andsometimes involves more complex linguistic analysis.

Input: "running", "runner", "ran" Lemmatization

Output: "run", "runner", "run"

Tools and Libraries for Stemming and Lemmatization

Several NLP libraries provide tools for stemming and lemmatization:

NLTK (Natural Language Toolkit): Provides implementations for both stemming and lemmatization.Example

(Stemming): nltk.stem.PorterStemmer, nltk.stem.SnowballStemmer

Example (Lemmatization): nltk.stem.WordNetLemmatizer

spaCy: An open-source library for advanced NLP in Python, with built-in lemmatization support.

Example: spacy.tokens.Token.lemma_

TextBlob: A simpler library for processing textual data, built on top of NLTK and Pattern.

Example (Lemmatization): textblob.Word.lemmatize

Stanford NLP: Provides robust tools for lemmatization as part of its suite of NLP tools.

Example: StanfordLemmatizer

When to Use Stemming vs. Lemmatization

Use Stemming: When speed and simplicity are more critical than accuracy. Suitable for applications like searchengines where the exact root form is not as important.

Use Lemmatization: When accuracy is crucial, and the proper base form of words is needed. Suitable for applications like machine translation, sentiment analysis, and any task requiring precise understanding of word forms.

Stemming and lemmatization are essential techniques in NLP for text normalization. While stemming is faster and simpler, it can be less accurate and produce non-word stems. Lemmatization, on the other hand, is more accurate and context-aware but requires more computational resources. The choice between stemming and lemmatization depends on the specific requirements and constraints of the task at hand.

✓ STEMMING

CODE:

```
import nltk
nltk.download('averaged_perceptron_tagger'
) from nltk.stem import PorterStemmer from
nltk.stem import SnowballStemmer from
nltk.stem import LancasterStemmer
words=['run','runner','running','ran','runs','easily','caring']



def portstemming(words):
   ps=PorterStemmer()
   print("Porter Stemmer") for
   word in words:
      print(word,'          >',ps.stem(word))



def snowballstemming(words):
   snowball=SnowballStemmer(language='english')
   print("Snowball Stemmer")
   for word in words:
      print(word,'          >',snowball.stem(word))



def lancasterstemming(words):
   lancaster=LancasterStemmer()
   print("Lancaster Stemmer") for
   word in words:
      print(word,'          >',lancaster.stem(word))

print("Select Operation.") print("1.Porter
Stemmer") print("2.Snowball Stemmer")
print("3.Lancaster
Stemmer")

while True:
   choice=input('Enter Choice(1/2/3):') if
   choice in ('1','2','3'):
      if choice == '1':
         print(portstemming(words))
      elif choice == '2':
         print(snowballstemming(words))
      elif choice == '3':
         print(lancasterstemming(words))
```

```
    next_calculation = input("Do you want to do stemming again? (yes/no):") if
    next_calculation== "no":


  break else:


    print("Invalid Input")
```

OUTPUT:

```
Select Operation.
1.Porter Stemmer
2.Snowball Stemmer
3.Lancaster Stemmer
Enter Choice(1/2/3):3
Lancaster Stemmer
run -----> run
runner -----> run
running -----> run
ran -----> ran
runs -----> run
easily -----> easy
caring -----> car
None
Do you want to do stemming again? (yes/no):no
```

CODE:

```
import nltk
from nltk.stem import WordNetLemmatizer
wordnet_lemmatizer = WordNetLemmatizer() text
=input("Enter words for Lemmatizing")
tokenization= nltk.word_tokenize(text)

# v verb, a adjective, n noun in lemmatize parameter for w
in tokenization:
   print("Lemma for {} is {}".format(w,wordnet_lemmatizer.lemmatize(w,'v'))) OUTPUT:
```

```
Enter words for Lemmatizing cries laughing walked
Lemma for cries is cry
Lemma for laughing is laugh
Lemma for walked is walk
```

Practical-3: Write a program to Implement a tri-gram model

Tri-gram model

A tri-gram model, also known as a trigram model, is a type of n-gram language model used in Natural Language Processing (NLP). It predicts the probability of a word given the two preceding words, effectively capturing the context provided by the previous two words. This is an extension of the bigram model, which considers only the previous word for prediction. Here's a detailed overview of the tri-gram model:

Key Concepts

N-Gram Model: An n-gram model predicts the probability of a word based on the previous

n−1 words. For a tri-gram model, n=3, so it predicts the current word based on the previous two words.

Markov Assumption: The tri-gram model relies on the Markov assumption, which states that the probability ofa word depends only on a fixed number of preceding words (in this case, two). This simplifies the modeling by ignoring the longer-range dependencies.

Probability Estimation

The probability of a word sequence

$w1, w2, \ldots,$n in a tri-gram model is approximated as:

$P(w1,w2,\ldots,wn) \approx P(w1) \cdot P(w2|w1) \cdot \prod i=3nP(wi|wi-2,wi-1)$

Here, $(wi|wi-2,wi-1)$Pis the conditional probability of word wi given the previous two words $wi-2$w and $wi-1$

Training a Tri-Gram Model

To train a tri-gram model, you need a large corpus of text to estimate the probabilities. The steps are:Corpus

Preparation: Collect a large, representative corpus of text.

Tokenization: Split the text into sentences and words.

Count Tri-Grams: Count the occurrences of each tri-gram (sequence of three words) in the corpus.

Calculate Probabilities: Calculate the conditional probabilities:

$P(wi|wi-2,wi-1)=C(wi-2,wi-1,wi)C(wi-2,wi-1)$Pw

Where

$C(wi-2,wi-1,wi)$Cw) is the count of the tri-gram $(wi-2,wi-1,wi)$(wi−2 ,wi−1 ,wi) and $C(wi-2,wi-1)$C(wi−2 ,wi−1) is the count of the bigram

$(wi-2,i-1)$(w i−2,wi−1).

Smoothing Techniques

In practice, many tri-grams will have zero counts, especially in smaller corpora. Smoothing techniques are usedto handle this issue:

Add-One (Laplace) Smoothing: Add one to all counts to ensure no zero probabilities.Good-

Turing Smoothing: Adjusts counts based on the frequency of frequencies.

Kneser-Ney Smoothing: A sophisticated method that adjusts the probability based on the number of contexts aword appears in, providing better estimates for rare tri-grams.

Applications of Tri-Gram Models

Tri-gram models are used in various NLP tasks, including:

Speech Recognition: Predicting the next word in a sequence to improve transcription accuracy.

Text Prediction: Suggesting the next word or phrase in text input applications.

Machine Translation: Improving the fluency of translated sentences by considering the context of the previoustwo words.

Consider the sentence: "The cat sat on the mat."

Tokenization: ["The", "cat", "sat", "on", "the", "mat"]

Tri-grams: [("The", "cat", "sat"), ("cat", "sat", "on"), ("sat", "on", "the"), ("on", "the", "mat")]

If you encounter a new context like "The cat", the tri-gram model can predict the next word by looking at theconditional probabilities derived from the training corpus.

The tri-gram model is a simple yet powerful tool in NLP that captures the context of the previous two words to predict the next word in a sequence. While it is limited by its fixed context window and the sparsity of data for higher-order n-grams, smoothing techniques and large corpora can mitigate these issues, making the tri-gram model a useful component in various language processing applications.

TEXT FILE:

```
Text summarization is the process of
creating a concise and fluent summary
of a longer text document.
```

CODE:

```
import nltk
nltk.download('punkt')
from nltk.tokenize import word_tokenize
from nltk import FreqDist
import pandas as pd

f = open("D:/salma/salma1.txt")
sample = f.read()

sample_tokens = nltk.word_tokenize(sample)
print('\n Sample Tokens:',sample_tokens)
print('\n Type of Sample Tokens:',type(sample_tokens)) print('\n
Length of Sample Tokens:',len(sample_tokens))

sample_freq =FreqDist(sample_tokens)
tokens=[]
sf=[]
for i in sample_freq:

    tokens.append(i)
    sf.append(sample_freq[i])

df = pd.DataFrame({'Tokens':tokens,'Frequency':sf})
print('\n',df)

print('\n Bigrams:',list(nltk.bigrams(sample_tokens))) print('\n
Trigrams:',list(nltk.trigrams(sample_tokens))) print('\n N-
grams(4):',list(nltk.ngrams(sample_tokens,4)))
```

OUTPUT:

```
 Sample Tokens: ['Text', 'summarization', 'is', 'the', 'process'
, 'of', 'creating', 'a', 'concise', 'and', 'fluent', 'summary',
'of', 'a', 'longer', 'text', 'document', '.']

 Type of Sample Tokens: <class 'list'>

 Length of Sample Tokens: 18

            Tokens  Frequency
0               of          2
1                a          2
2             Text          1
3   summarization          1
4               is          1
5              the          1
6          process          1
7         creating          1
8          concise          1
9              and          1
10          fluent          1
11         summary          1
12          longer          1
13            text          1
14        document          1
15               .          1
```

```
 Bigrams: [('Text', 'summarization'), ('summarization', 'is'), (
'is', 'the'), ('the', 'process'), ('process', 'of'), ('of', 'cre
ating'), ('creating', 'a'), ('a', 'concise'), ('concise', 'and')
, ('and', 'fluent'), ('fluent', 'summary'), ('summary', 'of'), (
'of', 'a'), ('a', 'longer'), ('longer', 'text'), ('text', 'docum
ent'), ('document', '.')]

 Trigrams: [('Text', 'summarization', 'is'), ('summarization', '
is', 'the'), ('is', 'the', 'process'), ('the', 'process', 'of'),
('process', 'of', 'creating'), ('of', 'creating', 'a'), ('creati
ng', 'a', 'concise'), ('a', 'concise', 'and'), ('concise', 'and'
, 'fluent'), ('and', 'fluent', 'summary'), ('fluent', 'summary',
'of'), ('summary', 'of', 'a'), ('of', 'a', 'longer'), ('a', 'lon
ger', 'text'), ('longer', 'text', 'document'), ('text', 'documen
t', '.')]

 N-grams(4): [('Text', 'summarization', 'is', 'the'), ('summariz
ation', 'is', 'the', 'process'), ('is', 'the', 'process', 'of'),
('the', 'process', 'of', 'creating'), ('process', 'of', 'creatin
g', 'a'), ('of', 'creating', 'a', 'concise'), ('creating', 'a',
'concise', 'and'), ('a', 'concise', 'and', 'fluent'), ('concise'
, 'and', 'fluent', 'summary'), ('and', 'fluent', 'summary', 'of'
), ('fluent', 'summary', 'of', 'a'), ('summary', 'of', 'a', 'lon
ger'), ('of', 'a', 'longer', 'text'), ('a', 'longer', 'text', 'd
ocument'), ('longer', 'text', 'document', '.')]
```

Practical-4: Write a program to Implement PoS tagging using HMM & Neural Model

PoS tagging using HMM & Neural Model

Part-of-Speech (PoS) tagging is a fundamental task in Natural Language Processing (NLP), where each word in a sentence is assigned its respective part of speech, such as noun, verb, adjective, etc. Two common approaches to PoS tagging are using Hidden Markov Models (HMM) and Neural Networks. Here's a detailed overview of both methods:

PoS Tagging using Hidden Markov Models (HMM)

An HMM is a statistical model that represents sequences with an underlying hidden process generating the observed sequence. In the context of PoS tagging, the observed sequence is the words in a sentence, and the hidden states are the PoS tags.

Components of HMM for PoS Tagging

States: The PoS tags (e.g., Noun, Verb, Adjective).

Observations: The words in the sentence.

Transition Probabilities (A): Probability of transitioning from one state (PoS tag) to another. **Emission**

Probabilities (B): Probability of a word being generated from a particular state (PoS tag).**Initial**

Probabilities (π): Probability of starting in each state (PoS tag).

Transition Probabilities: Estimated from the frequency of tag sequences in the training corpus.

Emission Probabilities: Estimated from the frequency of words associated with each tag in the training corpus.

Initial Probabilities: Estimated from the frequency of each tag at the beginning of sentences in the training corpus.

The Viterbi algorithm is used to find the most likely sequence of PoS tags for a given sentence. It uses dynamic programming to efficiently compute the most probable tag sequence.

Example: Suppose you have the sentence "The cat sat on the mat." The HMM would use transition and emission probabilities learned from a training corpus to find the most likely sequence of PoS tags.

PoS Tagging using Neural Networks

Neural networks, particularly Recurrent Neural Networks (RNNs) and their variants like Long Short-Term Memory (LSTM) networks and Bidirectional LSTMs (BiLSTMs), have become popular for sequence labeling tasks like PoS tagging due to their ability to capture context and long-range dependencies.

Neural Network Architecture for PoS Tagging

Embedding Layer: Converts words into dense vectors representing their meanings.

Recurrent Layer: Typically an LSTM or BiLSTM layer to capture sequential dependencies and context from both directions (forward and backward).

Dense Layer: Fully connected layer to map the output of the recurrent layer to the PoS tags.

Softmax Layer: Provides the probability distribution over PoS tags for each word.

The network is trained on a labeled corpus using backpropagation and an optimization algorithm (e.g., Adam). The objective is to minimize the categorical cross-entropy loss between the predicted tags and the true tags.

Given the sentence "The cat sat on the mat," the neural model would convert the words to embeddings, pass them through LSTM layers, and output a probability distribution over PoS tags for each word.

Comparison

Context Handling:

HMM: Limited to a fixed context (previous one or two tags).

Neural Networks: Can capture long-range dependencies and context through LSTM or BiLSTM layers.

Feature Engineering:

HMM: Requires manual feature engineering for emission probabilities.

Neural Networks: Automatically learn features from data through embeddings.

Performance:

HMM: Generally less accurate due to its limited context and reliance on pre-defined probabilities.

Neural Networks: Typically more accurate due to their ability to learn complex patterns and contextual information.

Complexity:

HMM: Simpler and easier to implement with smaller computational requirements.

Neural Networks: More complex with higher computational requirements but can leverage modern hardware accelerators like GPUs.

Conclusion

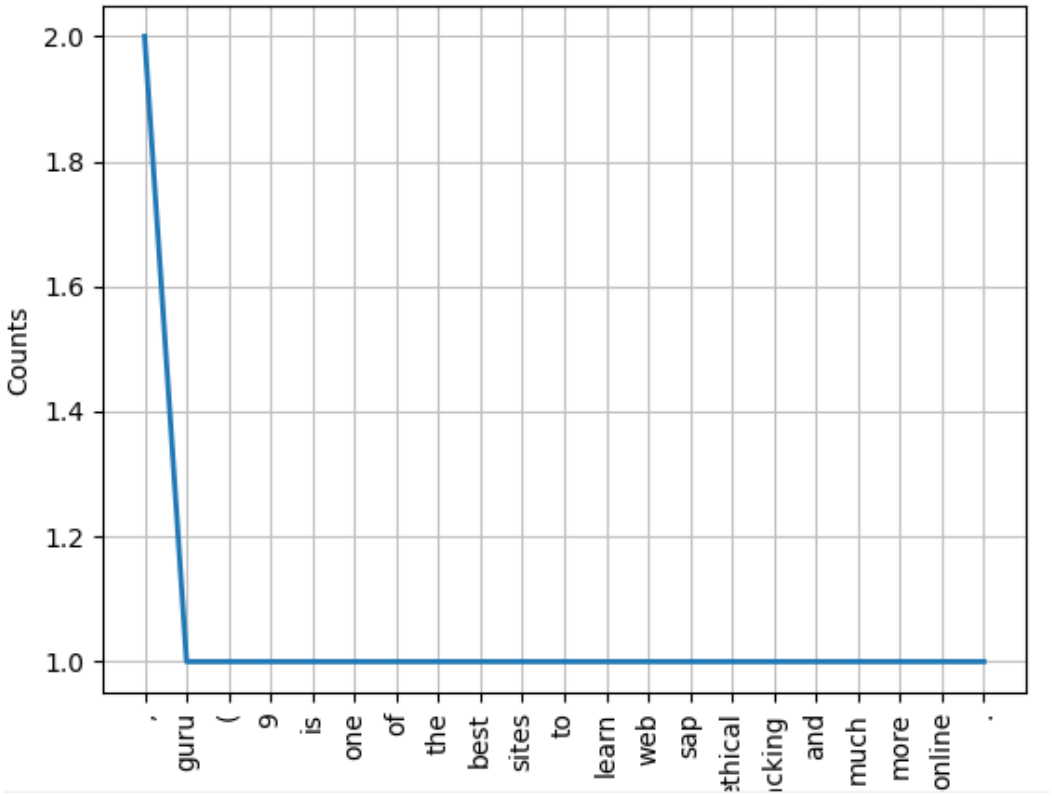
Both HMM and neural models have their own strengths and weaknesses. HMMs are simpler and computationally less intensive but lack the ability to capture long-range dependencies. Neural models, especially those using LSTMs or BiLSTMs, provide superior performance by leveraging context and learning complex patterns but require more computational resources and are more complex to implement. The choice of method depends on the specific requirements and constraints of the application.

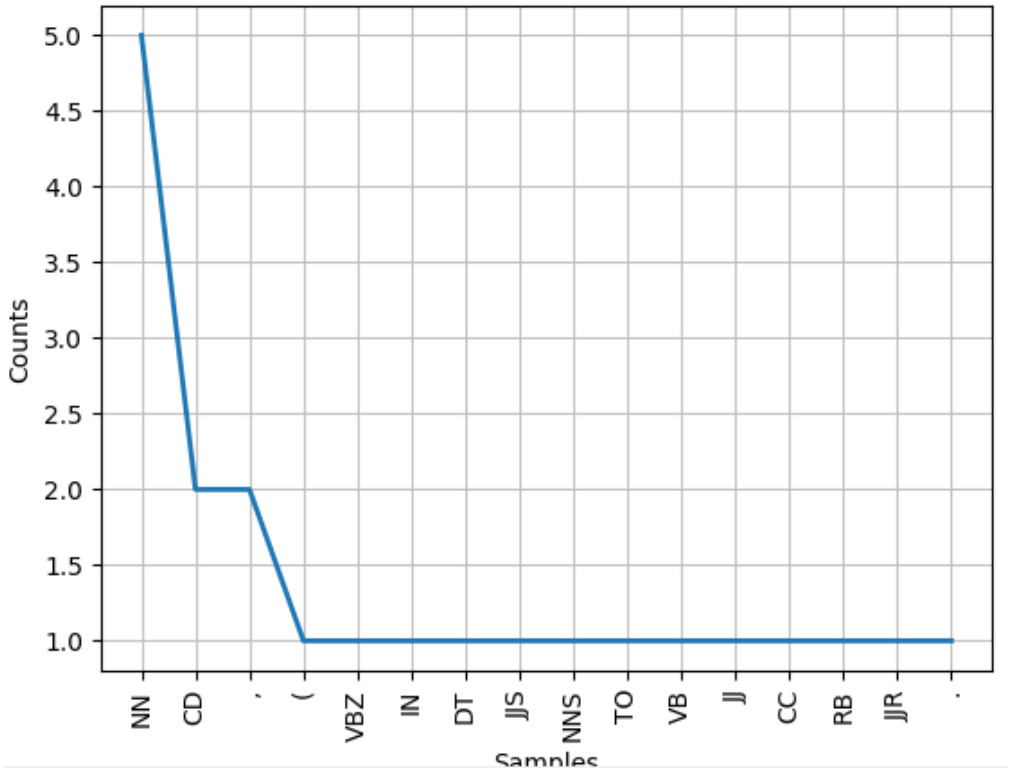
CODE:

```
import nltk
from collections import Counter

text="Guru(9 is one of the best sites to learn WEB,SAP,Ethical Hacking and much more online."
lower_case=text.lower()
tokens=nltk.word_tokenize(lower_case)
tags=nltk.pos_tag(tokens) print(tags)
counts=Counter(tag for word,tag in tags)


###for tag in tags: ###
print(tag)
print(counts)

fd=nltk.FreqDist(tokens)
fd.plot()


fd1=nltk.FreqDist(counts)
fd1.plot()
```

OUTPUT:

```
[('guru', 'NN'), ('(', '('), ('9', 'CD'), ('is', 'VBZ'), ('one', 'CD'), ('of', '
IN'), ('the', 'DT'), ('best', 'JJS'), ('sites', 'NNS'), ('to', 'TO'), ('learn',
'VB'), ('web', 'NN'), (',', ','), ('sap', 'NN'), (',', ','), ('ethical', 'JJ'),
('hacking', 'NN'), ('and', 'CC'), ('much', 'RB'), ('more', 'JJR'), ('online', 'N
N'), ('.', '.')]
Counter({'NN': 5, 'CD': 2, ',': 2, '(': 1, 'VBZ': 1, 'IN': 1, 'DT': 1, 'JJS': 1,
 'NNS': 1, 'TO': 1, 'VB': 1, 'JJ': 1, 'CC': 1, 'RB': 1, 'JJR': 1, '.': 1})
```


Counts
2.0
1.8
1.6
1.4
1.2
1.0
guru
(
9
is
one
of
the
best
sites
to
learn
web
sap
and
much
more
online


Counts
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
NN
CD
VBZ
IN
DT
JJS
NNS
TO
VB
JJ
CC
RB
JJR
Samples

Practical-5: Write a program to Implement syntactic parsing of a given text

Syntactic Parsing

Syntactic parsing, also known as syntactic analysis or parsing, is a fundamental task in Natural Language Processing (NLP) that involves analyzing the syntactic structure of sentences to identify their grammatical structure according to a given formal grammar. The goal is to create a parse tree or syntactic tree that representsthe hierarchical structure of the sentence, showing the relationships between words and phrases.

Types of Syntactic Parsing

Constituency Parsing (Phrase Structure Parsing):

Identifies the constituents (sub-phrases) of a sentence and their hierarchical structure.

Output: A constituency tree where each node represents a phrase or a word, and branches indicate the relationships between these phrases.

Example: For the sentence "The cat sat on the mat," a constituency tree might show "The cat" as a noun phrase (NP) and "sat on the mat" as a verb phrase (VP).

Dependency Parsing:

Focuses on the dependencies between words in a sentence, indicating which words depend on others.

Output: A dependency tree where nodes represent words, and directed edges represent dependencies between them.

Example: For the same sentence, the dependency tree would show "sat" as the root, with edges pointing to"cat," "on," and "mat," indicating their dependencies.

Approaches to Syntactic Parsing

Rule-Based Approaches:

Use manually crafted rules based on grammatical knowledge.

Can be precise for specific languages but lack scalability and flexibility.

Statistical Approaches:

Use probabilistic models trained on annotated corpora to predict the most likely parse tree for a sentence.

Examples include Probabilistic Context-Free Grammars (PCFGs).

Machine Learning Approaches:

Utilize supervised learning methods to train models on large annotated datasets.

Examples include neural network models such as RNNs, LSTMs, and Transformer-based models like BERT.Key

Algorithms and Models

Chart Parsing (CYK Algorithm):

Dynamic programming approach to parse sentences using context-free grammars.

Constructs a parse table (chart) to efficiently parse sentences by combining smaller sub-parses.

Earley Parser:

A versatile chart parsing algorithm that can handle any context-free grammar, including ambiguous and left-recursive grammars.

Uses a state machine and dynamic programming to efficiently parse sentences.

Transition-Based Dependency Parsing:

Constructs dependency trees by making a series of shift-reduce decisions.

Examples include the Arc-Standard and Arc-Eager algorithms.

Neural Network Models:

Constituency Parsing: Neural models like recursive neural networks (RecNNs) and neural chart parsers predict constituency trees.

Dependency Parsing: Neural models such as graph-based parsers (using biaffine classifiers) and transition-based parsers (using LSTMs and attention mechanisms).

Tools and Libraries

Stanford Parser: Provides both constituency and dependency parsers. Based on

statistical parsing models and can be used with Java and Python.

spaCy: A fast and efficient library for NLP, offering state-of-the-art dependency parsing.Supports

multiple languages and integrates well with other NLP tasks.

NLTK (Natural Language Toolkit):

Includes implementations for various parsing algorithms and grammars.Useful

for educational purposes and small-scale projects.

Applications of Syntactic Parsing

Machine Translation: Understanding sentence structure improves the quality of translations.

Information Extraction: Extracting structured information from unstructured text relies on parsing to identify relationships.

Question Answering: Parsing helps in understanding and extracting relevant information from text to answer questions accurately.

Text Summarization: Identifying key phrases and sentence structures aids in generating concise summaries.

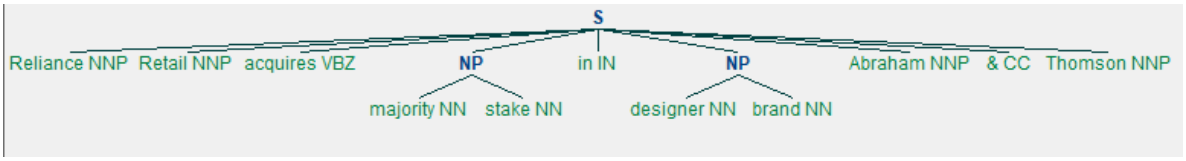
Sentiment Analysis: Parsing helps in identifying the syntactic structure that can influence sentiment analysis, such as negations.

Syntactic parsing is a critical component of NLP that provides a deeper understanding of sentence structure and grammar. Whether using rule-based, statistical, or machine learning approaches, syntactic parsers enable various downstream applications by converting raw text into structured, meaningful representations. The choiceof parsing technique depends on the specific requirements, language characteristics, and computational resources available for the task.

CODE:

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

from nltk import pos_tag,word_tokenize,RegexpParser


sentence='Reliance Retail acquires majority stake in designer brand Abraham & Thomson'
tokens=word_tokenize(sentence)
tags=pos_tag(tokens)
grammer="NP:{<NN>?<DT>*<NN>}"
chunker=RegexpParser(grammer)
result=chunker.parse(tags)
print(result)
result.draw()
```

```
(S
  Reliance/NNP
  Retail/NNP
  acquires/VBZ
  (NP majority/NN stake/NN)
  in/IN
  (NP designer/NN brand/NN)
  Abraham/NNP
  &/CC
  Thomson/NNP)
```


S
Reliance NNP
Retail NNP
acquires VBZ
NP
in IN
NP
Abraham NNP
& CC
Thomson NNP
majority NN
stake NN
designer NN
brand NN

Practical-6: Write a program to Implement dependency parsing of a given text

Dependency Parsing

Dependency parsing is a type of syntactic parsing that focuses on identifying the grammatical relationships between words in a sentence, represented as a dependency tree. In this tree, each node corresponds to a word, and edges represent syntactic dependencies between words. The goal is to capture which words depend on which other words and the nature of these dependencies.

Head and Dependent: In a dependency relation, one word acts as the head (or governor), and the other as the dependent (or modifier). For example, in the phrase "The cat sat," "sat" is the head, and "cat" is the dependent.

Root: The root of the dependency tree is usually the main verb or the core word of the sentence.

Labels: Edges in the dependency tree are often labeled with the type of grammatical relationship, such as subject, object, or modifier.

Types of Dependencies

Syntactic Dependencies: Relationships based on syntax, such as subject-verb, verb-object, and noun-modifier.

Semantic Dependencies: Relationships that capture the semantic roles of words, such as agent, patient, and instrument.

Dependency Parsing Methods

There are two main approaches to dependency parsing: transition-based and graph-based parsing.

Transition-Based Parsing

Concept: Builds the dependency tree incrementally by making a series of parsing decisions, such as attaching a word to its head or shifting the focus to the next word.

Algorithms: **Arc-Standard**: Builds the tree bottom-up by applying a series of shift and reduce operations.

Arc-Eager: Constructs the tree top-down, allowing for partial tree construction.

Given the sentence "The cat sat on the mat," a transition-based parser would start with the first word and incrementally decide where each subsequent word attaches.

Graph-Based Parsing

Concept: Considers all possible trees and scores them based on a global optimization criterion, typically finding the highest-scoring tree.

Algorithms: Eisner's Algorithm: A dynamic programming algorithm to find the optimal projective dependency tree.

Chu-Liu/Edmonds' Algorithm: Used for non-projective dependency trees.

For the same sentence "The cat sat on the mat," a graph-based parser would consider all possible tree structures and select the one with the highest score based on learned weights.

Neural Dependency Parsing

Recent advancements in neural networks have significantly improved the performance of dependency parsers.

Neural Network Models: Use deep learning techniques, such as Recurrent Neural Networks (RNNs), Long Short-Term Memory networks (LSTMs), and Transformers, to learn complex features and dependencies from large annotated corpora.

Embedding Layers: Convert words into dense vectors that capture syntactic and semantic properties. **BiLSTM**:

Often used to capture context from both directions (left-to-right and right-to-left) in a sentence.**Attention**

Mechanisms: Used to focus on relevant parts of the sentence when making parsing decisions.

Tools and Libraries for Dependency Parsing

spaCy: A fast and efficient library with state-of-the-art dependency parsing capabilities.

Stanford CoreNLP: Offers robust dependency parsers for multiple languages. **NLTK**:

Provides interfaces to various dependency parsers, including Stanford's.**Applications**

of Dependency Parsing

Information Extraction: Identifying relationships between entities in text.

Question Answering: Understanding the syntactic structure of questions and answers.

Machine Translation: Preserving syntactic relations in the translated text.

Text Summarization: Extracting key information based on syntactic importance.

Sentiment Analysis: Understanding how syntactic structures affect sentiment.

Example of Dependency Parsing

Consider the sentence: "The quick brown fox jumps over the lazy dog."

Sentence: "The quick brown fox jumps over the lazy dog."

Dependency Tree:

Root: "jumps"

"fox" → "jumps" (subject)

"The" → "fox" (determiner)

"quick" → "fox" (modifier)

"brown" → "fox" (modifier)

"over" → "jumps" (prepositional modifier)

"dog" → "over" (object of preposition) "the"

→ "dog" (determiner)

"lazy" → "dog" (modifier)

Dependency parsing is a powerful technique in NLP for understanding the grammatical structure and relationships in sentences. It provides a detailed representation of sentence structure, useful for various downstream applications. Both transition-based and graph-based methods, enhanced by neural network models, offer robust solutions for dependency parsing tasks.

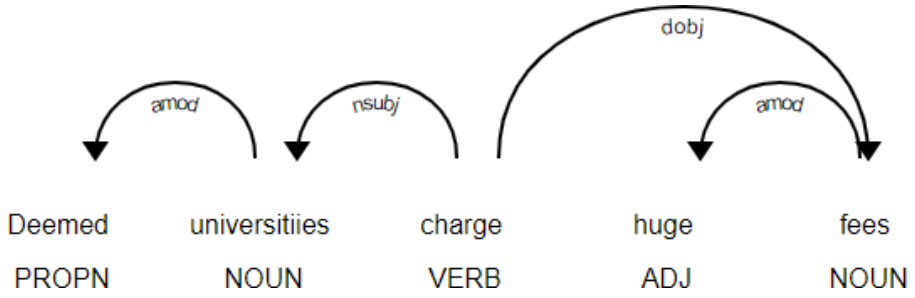
CODE:

```
#spacy download en_core_web_sm (install on cmd)
import spacy
from spacy import displacy

nlp = spacy.load("en_core_web_sm")
sentence = 'Deemed universitiies charge huge fees' doc
=nlp(sentence)
print("{:<15}|{:<8}|{:<15}|{:<20}".format('Token','Relation','Head','Children')) print('-
'*70)
for token in doc:
   print("{:15}|{:<8}|{:<15}|{:<20}".format(str(token.text),str(token.dep_),
                              str(token.head.text),str([child for child in token.children])))

# Use displacy to visualize the dependency
displacy.serve(doc,style='dep',options={'distance':120})
```

OUTPUT:

```
Token          |Relation|Head           |Children
----------------------------------------------------------------------
Deemed         |amod    |universitiies  |[]
universitiies  |nsubj   |charge         |[Deemed]
charge         |ROOT    |charge         |[universitiies, fees]
huge           |amod    |fees           |[]
fees           |dobj    |charge         |[huge]

Using the 'dep' visualizer
Serving on http://0.0.0.0:5000 ...

127.0.0.1 - - [14/Feb/2023 13:39:16] "GET / HTTP/1.1" 200 4233
127.0.0.1 - - [14/Feb/2023 13:39:16] "GET /favicon.ico HTTP/1.1" 200 4233
```

Copy the URL on web to see the dependency



Practical-7: Write a program to Implement Named Entity Recognition (NER)

Named Entity Recognition (NER)

Named Entity Recognition (NER) is a fundamental task in Natural Language Processing (NLP) that involves identifying and classifying named entities in text into predefined categories such as names of persons, organizations, locations, dates, times, quantities, monetary values, percentages, etc. The goal of NER is tolocate and classify these entities within a text accurately.

Entities: Items of interest in text that belong to predefined categories.

Examples:

Person: "Barack Obama"

Organization: "Google"

Location: "Paris"

Date: "June 1, 2024"

Categories: The predefined classes into which entities are classified.

Common categories include Person (PER), Organization (ORG), Location (LOC), Miscellaneous (MISC), Date (DATE), Time (TIME), Money (MONEY), and Percent (PERCENT).

Challenges:

Ambiguity: Words can have multiple meanings (e.g., "Apple" can be a fruit or a company). **Variability**:

Names can appear in various forms (e.g., "U.S.A." vs. "United States of America").**Context**: The context

in which a word appears can affect its classification.

Approaches to NER

Rule-Based Methods:

Use hand-crafted rules based on pattern matching and dictionaries to identify entities.Pros:

Simple to implement and understand.

Cons: Not scalable, high maintenance, and typically less accurate for complex texts.

Machine Learning-Based Methods:

Feature Engineering: Use features like word prefixes, suffixes, capitalization, part-of-speech tags, and contextwords.

Algorithms: Algorithms such as Conditional Random Fields (CRFs), Hidden Markov Models (HMMs), andSupport Vector Machines (SVMs).

Pros: More flexible and generally more accurate than rule-based methods.

Cons: Requires annotated training data and feature engineering.

Deep Learning-Based Methods:

Neural Networks: Use architectures such as Recurrent Neural Networks (RNNs), Long Short-Term Memorynetworks (LSTMs), and Transformers.

End-to-End Models: Learn features automatically from raw text data.

Contextual Embeddings: Use models like BERT, ELMo, and GPT to capture contextual word representations. Pros:

State-of-the-art performance, can capture complex patterns and dependencies.

Cons: Requires large annotated datasets and significant computational resources.

Popular NER Models and Tools

spaCy: A fast and efficient NLP library with a pre-trained NER model. Easy to use and integrate intoapplications.

Stanford NER: A Java-based NLP tool providing CRF-based NER. Supports multiple languages and is widelyused in academic research.

NLTK: Provides access to several NER tools, including Stanford NER, within the Python ecosystem.

Hugging Face Transformers: Provides implementations of Transformer-based models like BERT, RoBERTa,and GPT, with pre-trained models for NER.

Steps for Building an NER System

Data Collection and Annotation: Gather a large corpus of text and annotate it with named entity labels.

Preprocessing: Tokenize text, normalize case, and handle other text preprocessing tasks.

Feature Extraction (for traditional ML methods): Extract relevant features such as word shape, part-of-speech tags, and context words.

Model Training: Train the NER model using annotated data.

Traditional ML: Train using CRFs, HMMs, or SVMs.

Deep Learning: Train using RNNs, LSTMs, or Transformer-based models.

Evaluation: Evaluate the model using metrics such as Precision, Recall, and F1-Score on a held-out test set.

Deployment: Deploy the trained model in an application or service to perform NER on new text data.

Consider the sentence: "Barack Obama was born in Hawaii."

Tokenization: ["Barack", "Obama", "was", "born", "in", "Hawaii"]Entity

Recognition:

"Barack Obama" → Person (PER)

"Hawaii" → Location (LOC) Output:

[("Barack Obama", "PER"), ("Hawaii", "LOC")]

Applications of NER

Information Extraction: Automatically extracting relevant information from text documents.Search

Engines: Improving search accuracy by understanding the entities in queries.

Content Categorization: Automatically tagging content with relevant entities for better organization andretrieval.

Question Answering Systems: Understanding questions and extracting relevant entities to provide accurateanswers.

Customer Support: Identifying entities in customer queries to route them to the appropriate support channels.

NER is a critical component of many NLP applications, providing essential information about the entities mentioned in the text. Various approaches, from rule-based systems to advanced deep learning models, offer solutions for different use cases.

CODE:

```
import spacy
import pandas as pd
from spacy import displacy

NER = spacy.load("en_core_web_sm")
text = "Apple acquired zoom-in-China-on-wednesdav-6th May 2020.\ This news made Apple
and Google stock jump by 5% on Dow Jones Indexin -the \ United Starps of America"

doc = NER(text)
entities = [] labels
= [] position_start
= [] position_end =
[]

for ent in doc.ents:
  entities.append(ent)
  labels.append(ent.label_)
  position_start.append(ent.start_char)
  position_end.append(ent.end_char)
df = pd.DataFrame({'Entities': entities, 'Labels': labels, 'Position_Start': position_start, 'Position_End':
```

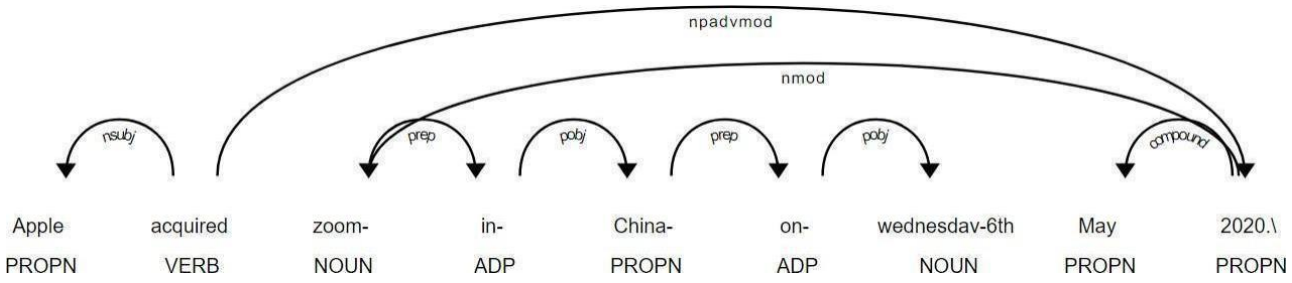
position_end})

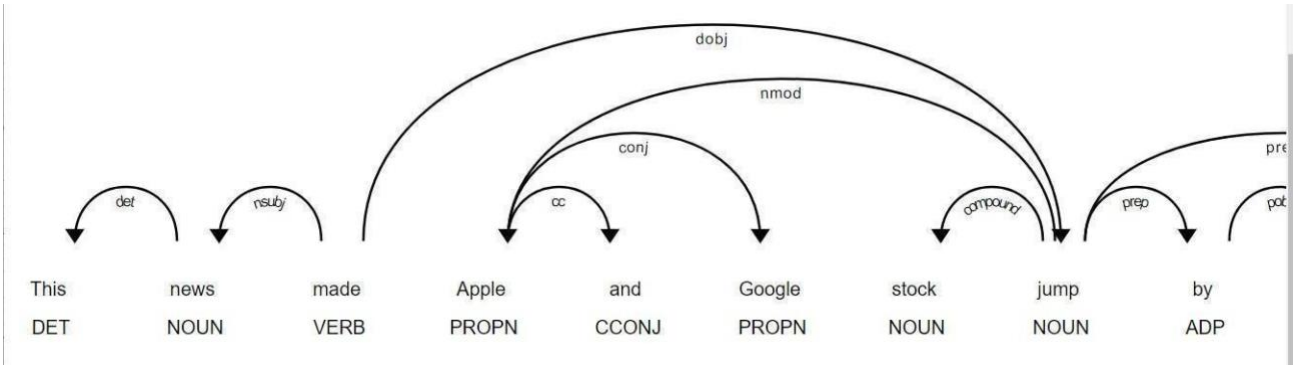
print(df)
displacy.serve(doc, style='dep', options={'distance': 120})
displacy.render(doc, style="ent")

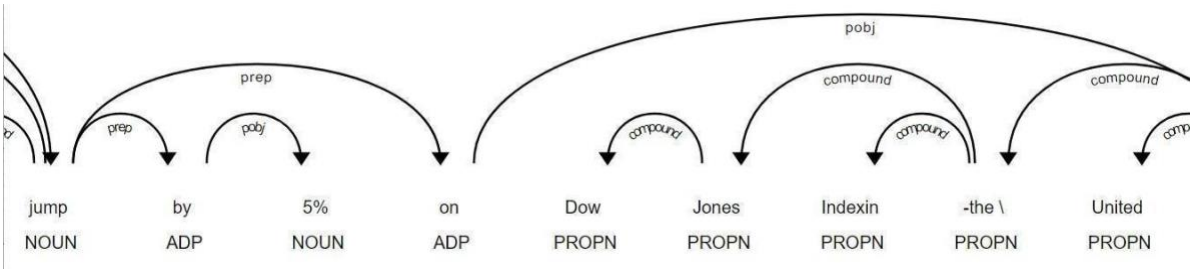
OUTPUT:

	Entities	Labels	Position_Start	Position_End
0	(Apple)	ORG	0	5
1	(China)	GPE	23	28
2	(May, 2020.\)	DATE	46	56
3	(Apple)	ORG	72	77
4	(Google)	ORG	82	88
5	(5, %)	PERCENT	103	105
6	(Dow, Jones, Indexin)	ORG	109	126
7	(United, Starps, of, America)	ORG	134	158

```
Using the 'dep' visualizer
Serving on http://0.0.0.0:5000 ...
```






pobj
prep
prep
pobj
compound
compound
compound
compound
jump
NOUN
by
ADP
5%
NOUN
on
ADP
Dow
PROPN
Jones
PROPN
Indexin
PROPN
-the \
PROPN
United
PROPN

Practical-8: Write a program to Implement Text Summarization for the given sample text

Text Summarization

Text summarization is the process of creating a concise and coherent version of a longer document while preserving its main points and overall meaning. It is a critical task in Natural Language Processing (NLP) with applications in various fields such as news aggregation, research paper summarization, and content curation.

Types of Text Summarization

Extractive Summarization:

Selects important sentences or phrases directly from the source text and concatenates them to form a summary.

Pros: Easier to implement, preserves original wording.

Cons: May lack coherence and fluency, depends on the quality of sentence selection.

Abstractive Summarization:

Generates a summary by interpreting and paraphrasing the content, producing novel sentences that may notappear in the original text.

Pros: Can produce more coherent and fluent summaries, captures the gist of the text better. Cons: More

complex to implement, requires advanced natural language generation techniques.Approaches to Text

Summarization

Rule-Based Methods:

Use predefined rules and heuristics to identify key sentences or phrases.

Techniques include sentence scoring based on features like word frequency, sentence position, and cue words.

Machine Learning-Based Methods:

Supervised Learning: Train models on labeled datasets where documents are paired with their corresponding summaries.

Unsupervised Learning: Use techniques like clustering to identify and extract key sentences without labeleddata.

Deep Learning-Based Methods:

Seq2Seq Models: Use encoder-decoder architectures, often with attention mechanisms, to generate summaries.

Transformer Models: Advanced models like BERT, GPT, and T5 that leverage large-scale pre-training andfine-tuning for summarization tasks.

Key Algorithms and Models

Latent Semantic Analysis (LSA): Unsupervised extractive method that uses singular value decomposition toidentify and extract important sentences.

TextRank: A graph-based ranking algorithm inspired by PageRank, where sentences are nodes, and edgesrepresent similarity scores. Sentences with higher ranks are selected for the summary.

Recurrent Neural Networks (RNNs): Used in early deep learning models for summarization. They capturesequential dependencies but struggle with long-term dependencies.

Long Short-Term Memory (LSTM) Networks: A type of RNN that mitigates the vanishing gradient problem,allowing for better handling of long sequences.

Attention Mechanism: Enhances seq2seq models by allowing the model to focus on different parts of the inputsequence when generating each word of the summary.

Transformers: Models like BERT and GPT use self-attention mechanisms to capture contextual informationmore effectively, enabling high-quality abstractive summarization.

Tools and Libraries

NLTK: Provides basic functionalities for text preprocessing and simple extractive summarization techniques.

Sumy: A Python library offering multiple summarization algorithms including LSA and TextRank.**Gensim**:

Includes an implementation of the TextRank algorithm for extractive summarization. **spaCy**: Can be used

for preprocessing and integrating with other summarization models.

Hugging Face Transformers: Offers pre-trained models for abstractive summarization, including BART, T5,and Pegasus.

Example of Summarization Consider

the following paragraph:

"Artificial intelligence (AI) refers to the simulation of human intelligence in machines. These machines are programmed to think like humans and mimic their actions. The term may also be applied to any machine that exhibits traits associated with a human mind such as learning and problem-solving. AI is continuously evolving to benefit many different industries."

Extractive Summary:

"Artificial intelligence (AI) refers to the simulation of human intelligence in machines. AI is continuously evolving to benefit many different industries."

Abstractive Summary:

"AI simulates human intelligence in machines, enabling learning and problem-solving. It's advancing across various industries."

Applications of Text Summarization

News Aggregation: Providing concise summaries of news articles.

Academic Research: Summarizing research papers to highlight key findings and contributions.

Legal Documents: Creating briefs and summaries of lengthy legal texts.

Customer Reviews: Summarizing customer feedback to extract common themes and sentiments.

Content Curation: Summarizing web content for newsletters, social media, and content recommendation systems.

Text summarization is a vital NLP task that helps in managing information overload by providing concise and relevant summaries. Extractive and abstractive summarization each have their strengths and suitable use cases. With advancements in deep learning, particularly with transformer models, abstractive summarization has seen significant improvements, offering more coherent and contextually accurate summaries. Tools like NLTK, spaCy, and Hugging Face Transformers make it easier to implement and experiment with different summarization techniques, enabling their application across various domains.

TEXT FILE:

```
NLP text summarization is the process of breaking down lengthy
text into digestible paragraphs or sentences. This method
extracts vital information while also preserving the meaning of
the text. This reduces the time required for grasping lengthy
pieces such as articles without losing vital information.

Text summarization is the process of creating a concise,
coherent, and fluent summary of a longer text document, which
involves underlining the document's key points.
```

CODE:

```
from nltk.tokenize import word_tokenize,sent_tokenize from
nltk.corpus import stopwords

f = open("D:/salma/salma.txt")
text = f.read()

words = word_tokenize(text)
sents = sent_tokenize(text)
stopwords = set(stopwords.words('english'))

freqTable = dict()
for word in words:
   word = word.lower() if
   word in stopwords:
```

```
        continue
    elif word in freqTable:
        freqTable[word]+=1
    else:
        freqTable[word]=1


sentValue = dict()
for sent in sents:
    for word,freq in freqTable.items(): if
        word in sent.lower():
            if sent in sentValue: sentValue[sent]+=freq
            else:
                sentValue[sent]=freq

sumValues = 0 for
s in sentValue:
    sumValues+=sentValue[s] avg =
int(sumValues/len(sents))

summary = "

for sent in sents:
    if (sent in sentValue) and (sentValue[sent]>1.2*avg):
        summary+="+sent
        print(summary)
```

OUTPUT:

```
Text summarization is the process of creating a
concise, coherent, and fluent summary of a longe
r text document, which involves underlining the
documentâ€™s key points.
```

Practical-9: Demonstrate the use of NLP in designing Virtual Assistants. Apply LSTM, build conversational Bots.

THEORY:

Chat – Chat is a class that contains complete logic for processing the text data which the chatbot receives and find useful information out of it.

Reflections – Another import we have done is reflections which is a dictionary containing basic input and corresponding outputs. You can also create your own dictionary with more responses you want. if you print reflections it will be something like this.

CODE:

```
import nltk

from nltk.chat.util import Chat

reflections = {

"i am" : "you are",

"i was" : "you were",

"i" : "you",

"i'm" : "you are",

"i'd" : "you would",

"i've" : "you have",

"i'll" : "you will",

"my" : "your",

"you are" : "I am",

"you were" : "I was",

"you've" : "I have",

"you'll" : "I will",
```

```
"your" : "my",
"yours" : "mine",
"you" : "me",
"me" : "you"
}

pairs = [
[
r"my name is (.*)",
["Hello %1, How are you today ?",]
],
[
r"hi|hey|hello",
["Hello", "Hey there",]
],
[
r"what is your name ?",
["I am a bot created by Analytics Vidhya. you can call me crazy!",]
],
[
r"how are you ?",
["I'm doing goodnHow about You ?",]
],
[
r"sorry (.*)",
["Its alright","Its OK, never mind",]
],
[
r"I am fine",
["Great to hear that, How can I help you?",]
```

```
],
[
r"i'm (.*) doing good",
["Nice to hear that","How can I help you?:)",]
],
[
r"(.*) age?",
["I'm a computer program dudenSeriously you are asking me this?",]
],
[
r"what (.*) want ?",
["Make me an offer I can't refuse",]
],
[
r"(.*) created ?",
["Raghav created me using Python's NLTK library ","top secret ;)",]
],
[
r"(.*) (location|city) ?",
['Indore, Madhya Pradesh',]
],
[
r"how is weather in (.*)?",
["Weather in %1 is awesome like always","Too hot man here in %1","Too cold man here
in %1","Never even heard about %1"]
],


[
r"i work in (.*)?",
["%1 is an Amazing company, I have heard about it. But they are in huge loss these
```

```
days.",]
],
[
r"(.*)raining in (.*)",
["No rain since last week here in %2","Damn its raining too much here in %2"]
],
[
r"how (.*) health(.*)",
["I'm a computer program, so I'm always healthy ",]
],
[
r"(.*) (sports|game) ?",
["I'm a very big fan of Football",]
],
[
r"who (.*) sportsperson ?",
["Messy","Ronaldo","Roony"]
],
[
r"who (.*) (moviestar|actor)?",
["Brad Pitt"]
],
[
r"i am looking for online guides and courses to learn data science, can you
suggest?",
["Crazy_Tech has many great articles with each step explanation along with code, you
can explore"]
],
[
r"quit",
```

```
["BBye take care. See you soon :) ","It was nice talking to you. See you soon :)"]

],

]

def chat():

  print("Hi! I am a chatbot created by Analytics Vidhya for your service")

  chat = Chat(pairs, reflections)

  chat.converse()

#initiate the conversation

if __name__ == "__main__":

          chat()
```

OUTPUT:

```
Hi! I am a chatbot created by Analytics Vidhya for your service
>hi
Hello
>how are you
I'm doing goodnHow about You ?
>nice
None
>
```