Practical No:1

Title:

Convert the text into tokens. Find the word frequency.

Aim:

To write a program that converts a given text into tokens and computes the frequency of each word.

Pre-requisites:

1. Understanding of Natural Language Processing (NLP) concepts, such as tokenization and word frequency.
2. Familiarity with basic string manipulation techniques in programming.
3. Knowledge of basic data structures like lists, dictionaries, and sets.
4. Understanding of how to handle text data, such as preprocessing and cleaning.

Theory:

Natural Language Processing (NLP) is a field of Artificial Intelligence (AI) that enables computers to understand, interpret, and manipulate human language. One of the foundational tasks in NLP is **tokenization** and **word frequency analysis**. These steps are essential for understanding the structure and meaning of the text.

1. **Tokenization:**
    - Tokenization is the process of breaking down text into smaller units called **tokens**. Tokens can be

words, sentences, or even characters, depending on the task.
    - **Word Tokenization** refers to splitting the text into individual words. This is typically done by splitting on whitespace and punctuation marks.
2. **Word Frequency:**
    - **Word frequency** refers to the number of times each word appears in a given text. Finding word frequency is useful in understanding the importance of words in a text, detecting common themes, and supporting further text analysis.
    - A common representation of word frequency is a dictionary where each unique word is a key, and its frequency is the value.

Steps for Tokenization and Word Frequency Calculation:

1. **Input Text:**
    - A string of text is provided as input.
2. **Text Preprocessing:**
    - **Lowercasing:** Convert all characters in the text to lowercase to ensure that word frequency is case-insensitive.
    - **Removing Punctuation:** Strip out punctuation marks such as periods, commas, and question marks to treat words with the same meaning equally (e.g., "word" and "word," are considered the same).
    - **Tokenization:** Split the preprocessed text into individual words.
3. **Word Frequency Calculation:**
    - Traverse the list of tokens.

- For each token, increment its count in a dictionary (or hash map) if it already exists, otherwise, initialize the count to 1.

4. **Output:**
   - Print the tokens.
   - Display the word frequency as a list of words with their corresponding counts.

Conclusion:

Tokenization and word frequency calculation are fundamental tasks in Natural Language Processing. Tokenization breaks the text into meaningful units, and word frequency helps to understand the distribution and importance of words. By preprocessing the text (lowercasing, removing punctuation), we ensure accurate results, regardless of variations in word forms. These tasks form the foundation for more advanced NLP techniques, such as sentiment analysis, topic modeling, and machine translation.

Marks			Subject Teacher Sign
U	P	A	

*Attach you practical code after this **(printed)**

Practical No:2

Title:

Find the synonym /antonym of a word using WordNet.

Aim:

To write a program that finds the synonyms and antonyms of a given word using the WordNet lexical database.

Pre-requisites:

1. Understanding of Natural Language Processing (NLP) concepts, particularly lexical semantics.
2. Familiarity with Python and libraries such as nltk (Natural Language Toolkit).
3. Knowledge of WordNet, a large lexical database of English words.
4. Basic understanding of how to use dictionaries and data structures to store word relationships.

Theory:

WordNet is a widely used lexical database in NLP that groups English words into sets of synonyms (called **synsets**) and provides short definitions, usage examples, and other linguistic relationships like antonyms (opposite meanings), hypernyms (broader categories), and hyponyms (subcategories).

Synonyms:

- Synonyms are words that have the same or nearly the same meaning as another word. Example: *happy* and *joyful* are synonyms.

Antonyms:

- Antonyms are words that have the opposite meaning to another word. Example: *happy* and *sad* are antonyms.

WordNet allows us to query these relationships programmatically. By using the `nltk` library in Python, we can access WordNet to find the synonyms and antonyms of words.

Steps for Finding Synonyms and Antonyms:

1. **Input Word:**
    - A word is provided as input.
2. **Using WordNet via nltk:**
    - Use WordNet's synsets to retrieve synonyms.
    - For each synonym, check if an antonym exists and retrieve it if available.
3. **Finding Synonyms:**
    - For the input word, find all synsets using `wordnet.synsets()`.
    - For each synset, retrieve its lemmas (words that are synonymous) using `synset.lemmas()`.
4. **Finding Antonyms:**
    - For each lemma, check if it contains an antonym by using `lemma.antonyms()`.
    - If antonyms are found, store them.
5. **Output:**
    - Display a list of synonyms and antonyms of the input word.

Conclusion:

Using WordNet through the `nltk` library, we can efficiently find synonyms and antonyms for any given word. Synonym and antonym

extraction is crucial in various NLP tasks, such as paraphrasing, sentiment analysis, and text generation. By leveraging WordNet's semantic relationships, we can better understand the meaning and context of words, making it an essential tool for many language processing applications.

Marks			Subject Teacher Sign
U	P	A	

*Attach you practical code after this **(printed)**

Practical No:3

Title:

Demonstrate a bigram / trigram language model. Generate regular expressions for a given text.

Aim:

1. To demonstrate how to build bigram and trigram language models.
2. To generate a regular expression for a given text.

Pre-requisites:

1. Basic understanding of NLP concepts, especially **n-gram models**.
2. Familiarity with probability theory and how n-gram models are used in predicting sequences.
3. Knowledge of regular expressions and their role in pattern matching.
4. Familiarity with Python and NLP libraries such as `nltk`.

Theory:

Part 1: N-gram Language Models (Bigram and Trigram)

An **N-gram model** is a type of probabilistic language model used to predict the next word in a sequence, given the previous N-1 words. It assumes that the probability of a word depends only on the previous words (up to N-1).

- **Unigram Model:** The probability of each word depends only on itself.
- **Bigram Model (2-gram):** The probability of a word depends on the previous word.

    - Example: In the sentence, “I love coding,” a bigram model breaks it into pairs: (I, love), (love, coding).
- **Trigram Model (3-gram):** The probability of a word depends on the previous two words.
    - Example: “I love coding” is broken into trigrams: (I, love, coding).

Part 2: Regular Expressions

A **regular expression (regex)** is a sequence of characters that define a search pattern. Regular expressions are used for string matching, text search, and text manipulation.

Common regex patterns include:

- **Character Classes:** [a-zA-Z] matches any letter.
- **Quantifiers:** * (0 or more), + (1 or more), ? (0 or 1).
- **Anchors:** ^ (start of string), $ (end of string).

Steps for Bigram/Trigram Model:

1. **Text Preprocessing:**
    - Convert the text to lowercase.
    - Tokenize the text into words.
    - Remove stop words or punctuation (optional).
2. **Build the Bigram/Trigram Model:**
    - Use a bigram or trigram model to calculate word pair or triplet frequencies.
    - Calculate conditional probabilities of words appearing after one or two words.
3. **Generate Sentence:**

- Starting with a word, use the bigram or trigram model to predict the next word based on probability.

Steps for Regular Expression Generation:

1. **Input a Text:**
   - Extract key patterns, such as email addresses, phone numbers, or specific word sequences.
2. **Create Regex Pattern:**
   - Write regular expressions that match specific patterns in the text.

Conclusion:

Bigram and trigram models are important for capturing the local dependencies between words in text, making them useful for tasks like sentence generation or next-word prediction. Regular expressions are powerful tools for extracting patterns such as email addresses or phone numbers from text, making them invaluable in text processing tasks. Together, these techniques are foundational in many NLP applications.

Marks			Subject Teacher Sign
U	P	A	

*Attach you practical code after this **(printed)**

Practical No:4

Title:

Perform Lemmatization and Stemming. Identify parts-of Speech using Penn Treebank tag set.

Aim:

1. To perform **lemmatization** and **stemming** on a given text.
2. To identify and tag parts of speech (POS) using the **Penn Treebank Tag Set**.

Pre-requisites:

1. Understanding of Natural Language Processing (NLP) techniques, specifically **lemmatization**, **stemming**, and **POS tagging**.
2. Familiarity with libraries such as `nltk` for NLP tasks.
3. Knowledge of the **Penn Treebank Tag Set**, which is a standard POS tagging system used in English corpora.

Theory:

Lemmatization:

- **Lemmatization** is the process of reducing a word to its base or dictionary form, known as the **lemma**. Lemmatization considers the **context** and **part of speech** of the word, which results in more meaningful reductions compared to stemming.
    - Example: The word *running* becomes *run*, and *better* becomes *good*.

Stemming:

- **Stemming** is a more rudimentary process of chopping off the end of words to reduce them to their root forms (known as **stems**). Stemming does not consider the context of the word and often results in non-real words.
    - Example: The word *running* becomes *run*, and *studies* becomes *studi*.

Part-of-Speech (POS) Tagging:

- POS tagging is the process of assigning a tag to each word in a sentence that corresponds to its grammatical category. These categories include nouns, verbs, adjectives, etc.
- **Penn Treebank Tag Set** is a commonly used POS tag set that includes 36 POS tags, such as:
    - **NN**: Noun, singular
    - **VB**: Verb, base form
    - **JJ**: Adjective
    - **RB**: Adverb
    - **PRP**: Personal pronoun

Steps for Lemmatization and Stemming:

1. **Text Preprocessing:**
    - Input text is tokenized into individual words.
    - Convert the text into lowercase for uniformity.
2. **Perform Lemmatization:**
    - Use lemmatization to convert each word into its base form using a lemmatizer like `WordNetLemmatizer`.
3. **Perform Stemming:**

- Apply stemming to each word using a stemmer like `PorterStemmer`.

Steps for POS Tagging Using Penn Treebank Tag Set:

1. **Text Preprocessing:**
   - Tokenize the sentence into words.
2. **POS Tagging:**
   - Use a POS tagger from `nltk` to assign parts-of-speech tags to each word according to the Penn Treebank Tag Set.
3. **Output Tags:**
   - Display the tagged sentence with the POS categories of each word.

Conclusion:

Lemmatization and stemming are essential techniques in text normalization for many NLP applications. Lemmatization produces linguistically meaningful base forms, while stemming reduces words more crudely to their stems. POS tagging, using the Penn Treebank Tag Set, provides grammatical information about each word in a sentence, making it crucial for tasks like syntactic parsing, named entity recognition, and machine translation. These processes are foundational steps in building sophisticated language models.

Marks			Subject Teacher Sign
U	P	A	

*Attach you practical code after this **(printed)**

Practical No:5

Title:

Implement HMM for POS tagging. Build a Chunker

Aim:

1. To implement a **Hidden Markov Model (HMM)** for Part-of-Speech (POS) tagging.
2. To build a **Chunker** for identifying syntactic structures such as noun phrases and verb phrases.

Pre-requisites:

1. Understanding of **Hidden Markov Models (HMM)** and their applications in sequence modeling.
2. Familiarity with **Part-of-Speech tagging** and how POS tags help in identifying syntactic structures.
3. Understanding of **Chunking**, the process of extracting short phrases from a sentence.
4. Knowledge of NLP libraries like `nltk` to implement POS tagging and chunking.

Theory:

1. Hidden Markov Model (HMM) for POS Tagging

A **Hidden Markov Model (HMM)** is a probabilistic model used to represent a sequence of observed events, such as words, based on hidden states, such as part-of-speech tags. HMM assumes that:

- The sequence of words depends on a hidden sequence of POS tags.

- Each word in the sentence is emitted from a corresponding POS tag.

2. Chunking (Shallow Parsing)

Chunking is the process of identifying short phrases in a sentence, such as noun phrases (NP), verb phrases (VP), etc., by grouping POS tags. The process is often referred to as shallow parsing because it does not provide the deep hierarchical structure of a full syntactic parse.

- Noun Phrase (NP): A chunk consisting of a noun and any preceding adjectives or determiners (e.g., "the big dog").
- Verb Phrase (VP): A chunk consisting of a verb and any associated objects or complements (e.g., "is running quickly").

In chunking, regular expressions can be defined based on POS tag sequences to extract these chunks from the sentence.

Steps for HMM POS Tagging:

1. Text Preprocessing:
    - Tokenize the sentence into words.
2. HMM Training:
    - Train the HMM on a tagged corpus to estimate transition probabilities between POS tags and emission probabilities from POS tags to words.
3. POS Tagging:
    - Use the Viterbi algorithm to predict the most likely sequence of POS tags for a given sentence.

Steps for Chunker:

1. Text Preprocessing:
    - Tokenize the sentence into words and apply POS tagging.
2. Chunking:
    - Define regular expression patterns to extract noun and verb phrases based on the POS tag sequences.
3. Chunk Extraction:
    - Apply the chunking patterns to extract chunks such as noun phrases and verb phrases.

Conclusion:

Implementing **HMM for POS tagging** is useful for predicting the most likely sequence of tags for a sentence, especially when the sequence of words is known, but the tags are hidden. **Chunking** helps in extracting useful syntactic structures like noun and verb phrases, which can be used in further tasks such as named entity recognition (NER) and shallow parsing. Together, these techniques provide foundational tools for syntactic analysis in NLP tasks.

Marks			Subject Teacher Sign
U	P	A	

*Attach you practical code after this **(printed)**

Practical No:6

Title:

Implement Named Entity Recognizer.

Aim:

To implement a **Named Entity Recognizer (NER)** to identify and classify named entities in a given text, such as **person names**, **organizations**, **locations**, **dates**, etc.

Pre-requisites:

1. Basic understanding of **Named Entity Recognition (NER)**.
2. Familiarity with NLP libraries such as `nltk` or `spaCy` for NER.
3. Knowledge of machine learning models used in NER, like Conditional Random Fields (CRF) or pre-trained models for entity recognition.

Theory:

Named Entity Recognition (NER):

NER is the process of locating and classifying named entities in unstructured text into predefined categories, such as:

- **PERSON**: Names of people (e.g., "Albert Einstein").
- **ORGANIZATION**: Names of organizations (e.g., "Google").
- **LOCATION**: Geographic locations (e.g., "New York").
- **DATE/TIME**: Dates or times (e.g., "January 1st, 2024").
- **MONEY**: Monetary values (e.g., "$10 million").
- **PERCENTAGE**: Percentages (e.g., "30%").

NER is widely used in various NLP tasks such as information retrieval, question answering, and summarization.

How NER Works:

1. **Text Preprocessing:** Tokenize the text into sentences and words.
2. **POS Tagging:** Assign Part-of-Speech (POS) tags to words to provide syntactic information.
3. **Entity Recognition:** Use NER models to recognize named entities in the text and classify them into predefined categories.
4. **Contextual Understanding:** Some NER systems leverage pre-trained models that understand the context of a word in a sentence to better classify the entities.

Steps for Named Entity Recognition:

1. **Text Preprocessing:**
    - Tokenize the input text into sentences and words.
2. **POS Tagging (Optional):**
    - Use a POS tagger to assign grammatical categories to each word (e.g., noun, verb).
3. **Named Entity Recognition:**
    - Apply an NER model, such as one available in the `nltk` or `spaCy` libraries, to recognize named entities.
4. **Classify Named Entities:**
    - Entities are classified into categories like PERSON, ORGANIZATION, LOCATION, etc.

Conclusion:

Named Entity Recognition (NER) is a crucial task in natural language processing that enables systems to identify and classify important pieces of information from unstructured text. Implementing NER using libraries like nltk and spaCy simplifies the process as they provide pre-trained models for entity recognition. This technique is widely used in applications like search engines, recommendation systems, and information extraction.

Marks			Subject Teacher Sign
U	P	A	

*Attach you practical code after this **(printed)**

Practical No:7

Title:
Implement Semantic Role Labeling (SRL) to Identify Named Entities

Aim:
To implement Semantic Role Labeling (SRL) and use it to identify Named Entities (NE) within a sentence and assign semantic roles to these entities based on their functions in the sentence.

Pre-requisites:

1. Understanding of Semantic Role Labeling (SRL), its purpose in NLP, and how it assigns roles to sentence constituents.
2. Familiarity with Named Entity Recognition (NER) to identify entities like persons, organizations, and locations.
3. Knowledge of Python libraries, particularly `nltk` and `spaCy` for NLP tasks.

Theory:

Semantic Role Labeling (SRL):

SRL is a task in Natural Language Processing (NLP) that assigns roles to words or phrases in a sentence. These roles represent the semantic relationships between the predicate (often a verb) and its arguments (the entities involved in the action). For example, in the sentence:

> "John gave Mary a book."

- John is the Agent (the one performing the action),
- Mary is the Recipient (the one receiving something),
- a book is the Theme (the object being acted upon).

Semantic Roles:

1. **Agent:** The doer of the action.
2. **Patient/Theme:** The entity affected by the action.
3. **Instrument**: The tool used to perform the action.
4. **Location:** Where the action takes place.
5. **Recipient**: The entity that receives something.

By identifying these roles, SRL can provide a deeper understanding of the meaning of a sentence.

Steps for Semantic Role Labeling:

1. **Text Preprocessing:**
    - Tokenize the input text into sentences and words.
2. **POS Tagging and Dependency Parsing:**
    - Perform POS tagging to assign syntactic roles to each word.
    - Parse the sentence to establish dependencies between the words, identifying the predicate (main verb).
3. **Semantic Role Labeling:**
    - Identify the predicate (verb) in the sentence.
    - Assign semantic roles to each of the arguments related to the predicate.
4. **Named Entity Recognition:**
    - Apply NER to the same text to recognize named entities such as PERSON, ORGANIZATION, and LOCATION.

- Combine NER with SRL to link named entities with their roles in the sentence.

Conclusion:

Implementing Semantic Role Labeling (SRL) along with Named Entity Recognition (NER) provides a powerful approach to understanding the meaning of a sentence. SRL identifies the roles of entities in relation to the actions or events described in the sentence, while NER classifies named entities like people, organizations, and locations. Together, these techniques offer enhanced capabilities for semantic analysis and can be used in a variety of NLP applications.

Marks			Subject Teacher Sign
U	P	A	

*Attach you practical code after this **(printed)**

Practical No:8

Title:

Implement text classifier using logistic regression model

Aim:

To implement a text classifier using logistic regression, which can classify text data into distinct categories based on the provided input features.

Pre-requisites:

1. Understanding of machine learning concepts, especially classification.
2. Familiarity with logistic regression and its applications in binary and multi-class classification.
3. Basic knowledge of text preprocessing techniques (tokenization, vectorization, etc.).
4. Experience with Python libraries such as scikit-learn for model building and pandas for data handling.

Theory:

Text classification is a natural language processing (NLP) task where text data is categorized into predefined labels. A logistic regression model, a linear model widely used for binary classification, estimates the probability of a text sample belonging to a particular class based on feature input.

Logistic Regression

Logistic regression is a supervised learning algorithm that uses a linear combination of features, applies the logistic function, and outputs a probability between 0 and 1. It's

particularly effective in binary classification tasks but can also handle multi-class classification using techniques such as one-vs-rest.

Given features X and a binary target variable y, the logistic regression model is defined as:

$$P(y = 1|X) = \frac{1}{1 + e^{-(wX+b)}}$$

where:

- w is the weight vector,
- b is the bias term, and
- e is the base of the natural logarithm.

Steps to Implement a Text Classifier with Logistic Regression:

1. Data Collection: Use a labeled dataset for text classification (e.g., spam vs. not spam, sentiment analysis).
2. Text Preprocessing:
    - Tokenization: Split the text into words (tokens).
    - Stop Word Removal: Remove commonly used words that may not add meaningful information.
    - Vectorization: Convert tokens to numerical features using techniques like Bag of Words or TF-IDF (Term Frequency-Inverse Document Frequency).
3. Train-Test Split: Divide the dataset into training and testing sets to evaluate the model.
4. Model Building: Train the logistic regression model on the vectorized features.

5. Model Evaluation: Evaluate the model's accuracy, precision, recall, and F1-score.

Conclusion:

Using logistic regression for text classification is efficient and provides robust results for binary classification tasks. This implementation demonstrates basic NLP preprocessing steps like tokenization and TF-IDF vectorization, which are essential for converting text data into numerical form. The logistic regression model successfully learns from these features to classify text data into predefined categories.

Marks			Subject Teacher Sign
U	P	A	

*Attach you practical code after this **(printed)**

Practical No:9

Title:

Implement a movie reviews sentiment classifier

Aim:

To implement a **sentiment classifier** for movie reviews using natural language processing techniques and a machine learning model (e.g., **Logistic Regression**) to classify reviews as either **positive** or **negative**.

Pre-requisites:

1. Basic understanding of **sentiment analysis** and its applications.
2. Knowledge of text preprocessing techniques like tokenization, removing stopwords, and feature extraction (TF-IDF, Bag of Words).
3. Familiarity with machine learning models such as **Logistic Regression**, **Naive Bayes**, or **SVM**.
4. Python libraries such as `scikit-learn` for machine learning and `nltk` for text processing.

Theory:

Sentiment Analysis:

Sentiment Analysis is the process of determining the sentiment (positive, negative, or neutral) behind a piece of text. In this case, we are analyzing **movie reviews** to predict whether they are positive or negative based on the textual content.

Approach:

1. **Text Preprocessing:**
    - Tokenize the text, remove stopwords, perform stemming/lemmatization, and convert text into a numerical form using **Bag of Words** or **TF-IDF**.
2. **Model Selection:**
    - Use a supervised learning algorithm such as **Logistic Regression** for binary classification (positive vs. negative).
3. **Feature Extraction:**
    - Extract features from text data using methods like **TF-IDF** (Term Frequency-Inverse Document Frequency).
4. **Evaluation:**
    - Evaluate the performance of the sentiment classifier using metrics like **accuracy**, **precision**, **recall**, and **F1-score**.

Steps for Movie Review Sentiment Classification:

1. **Text Preprocessing:**
    - Tokenize and clean the text (remove punctuation, stopwords, etc.).
    - Apply stemming or lemmatization to normalize the text.
2. **Feature Extraction:**
    - Use **TF-IDF Vectorizer** to transform the text into feature vectors.
3. **Model Training:**
    - Train a **Logistic Regression** model on labeled movie reviews (positive and negative).
4. **Model Evaluation:**

- Evaluate the trained model on test data using performance metrics like accuracy.

Conclusion:

In this assignment, we successfully implemented a movie reviews sentiment classifier using the Logistic Regression model. The process involved text preprocessing, feature extraction using TF-IDF Vectorizer, and training a classifier to predict whether a review is positive or negative. This approach provides a foundation for sentiment analysis in real-world applications, such as customer reviews and social media analysis.

Marks			Subject Teacher Sign
U	P	A	

*Attach you practical code after this **(printed)**

Practical No:10

Title:

Implement RNN for sequence labelling

Aim:

To implement a **Recurrent Neural Network (RNN)** for sequence labeling tasks, such as part-of-speech tagging or named entity recognition.

Pre-requisites:

1. Basic understanding of **Recurrent Neural Networks (RNNs)** and their applications in sequence data.
2. Familiarity with Python libraries such as TensorFlow or PyTorch for building neural networks.
3. Knowledge of preprocessing techniques, especially for text data (tokenization, padding).
4. Understanding of sequence labeling tasks and evaluation metrics.

Theory:

Recurrent Neural Networks (RNNs):

RNNs are a class of neural networks designed to process sequential data. Unlike traditional feedforward networks, RNNs have connections that allow information to persist across time steps, making them suitable for tasks where the context of previous elements influences the current prediction (e.g., text).

Applications of RNNs:

- Part-of-Speech (POS) tagging
- Named Entity Recognition (NER)
- Machine translation
- Speech recognition

Sequence Labeling:

Sequence labeling involves assigning a label to each element in a sequence. For instance, in POS tagging, each word in a sentence is assigned a grammatical category (e.g., noun, verb). In NER, entities are tagged with categories like PERSON, ORGANIZATION, or LOCATION.

Architecture:

An RNN consists of an input layer, recurrent hidden layers, and an output layer. The hidden layers maintain a hidden state that is updated at each time step based on the current input and the previous hidden state.

Steps for Implementing RNN for Sequence Labeling:

1. **Data Preparation:**
    - Load and preprocess the dataset (tokenization, padding).
    - Create input-output pairs for the RNN (e.g., words and their corresponding labels).
2. **Build the RNN Model:**
    - Construct the RNN architecture using a framework like TensorFlow or PyTorch.
    - Use techniques like **embedding layers** for word representations.

3. **Training the Model:**
    - Compile the model with an appropriate loss function and optimizer.
    - Train the model on the training data.
4. **Model Evaluation:**
    - Evaluate the model on test data and calculate metrics such as accuracy and F1-score.

Conclusion:

In this assignment, we successfully implemented an RNN for a sequence labeling task such as part-of-speech tagging using a sample dataset. The model was trained using the LSTM architecture, which is an effective variant of RNNs for handling sequences. This approach can be adapted for various sequence labeling tasks, including named entity recognition and other NLP applications.

Marks			Subject Teacher Sign
U	P	A	

*Attach you practical code after this **(printed)**