LAB – 1 : WORD ANALYSIS

AIM: To perform basic word analysis

ALGORITHM:

1. Import spaCy: Import the spaCy library, which is an open-source natural language processing library.

2. Load Language Model: Load the English language model (en_core_web_sm) provided by spaCy, which includes a tokenizer, tagger, parser, named entity recognizer (NER),and word vectors.

3. Define Sample Text: Set a sample text that will be used for analysis. In this case, it's

"Natural Language Processing is a fascinating field of study."

4. Process Text with spaCy: Pass the sample text through the loaded spaCy pipeline using nlp(text). This processes the text and generates a doc object that contains linguistic annotations and information about the text.

5. Tokenization and Lemmatization: Extract tokens: Get a list of all the tokens (individual words) in the text using a list

comprehension [token.text for token in doc].

Lemmatization: Get a list of lemmas (base forms of words) using [token.lemma_ for token in doc].

6. Print Tokens and Lemmas: Print the extracted tokens and their corresponding lemmas.

7. Dependency Parsing: Iterate through each token in the processed doc.Print information about each token's text, dependency relation (token.dep_), its head text (token.head.text), its head's part-of-speech (token.head.pos_), and its children ([child for child in token.children]).

PROGRAM

```
import spacy

# Load English tokenizer, tagger, parser, NER, and word vectors

nlp = spacy.load("en_core_web_sm")

# Sample text for analysis

text = "Natural Language Processing is a fascinating field of study."

# Process the text with spaCy

doc = nlp(text)

# Extracting tokens and lemmatization

tokens = [token.text for token in doc]

lemmas = [token.lemma_ for token in doc]

print("Tokens:", tokens)

print("Lemmas:", lemmas)

# Dependency parsing

print("\nDependency Parsing:")

for token in doc:

        print(token.text, token.dep_, token.head.text, token.head.pos_,

        [child for child in token.children])
```

OUTPUT:

```
Tokens: ['Natural', 'Language', 'Processing', 'is', 'a', 'fascinating', 'field', 'of', 'study', '.']

Lemmas: ['Natural', 'Language', 'Processing', 'be', 'a', 'fascinating', 'field', 'of', 'study', '.']

Dependency Parsing:

Natural compound Language PROPN []

Language compound Processing PROPN [Natural]

Processing nsubj is AUX [Language]

is ROOT is AUX [Processing, field, .]

a det field NOUN []

fascinating amod field NOUN []

field attr is AUX [a, fascinating, of]

of prep field NOUN [study]

study pobj of ADP []

. punct is AUX []
```

LAB 2: WORD GENERATION

AIM: Word generation using NLTK

ALGORITHM:

```
1. Install and Import Libraries: Install NLTK (!pip install nltk) and import the required libraries (nltk and random).
2. Download NLTK Resources: Download NLTK resources, specifically the 'punkt' and 'gutenberg' corpora.
3. Load Corpus: Load a corpus from NLTK (e.g., Gutenberg corpus).
4. Create Bigram Model: Generate a list of bigrams (pairs of consecutive words) from the loaded corpus.
5. Choose Starting Word: Select a starting word for text generation.
6. Generate Text:
        a. Iterate a specified number of times (here, 20 iterations).
        b. For each iteration:
        c. Find all possible words that follow the last generated word in the bigrams.
        d. Randomly select a word from the possible words to continue the sequence.
        e. Append the selected word to the generated text.
7. Output Generated Text: Display the generated text.
```

PROGRAM:

```
!pip install nltk

import nltk

import random
```

```
# Download NLTK resources (run only once if not downloaded)
nltk.download('punkt')
nltk.download('gutenberg')
# Load a corpus (for example, the Gutenberg corpus)
words = nltk.corpus.gutenberg.words()
# Create a bigram model
bigrams = list(nltk.bigrams(words))
# Choose a starting word (you can choose any word from the corpus)
starting_word = "the"
generated_text = [starting_word]
# Generate 20 words of text
for _ in range(20):
 # Get all bigrams that start with the last generated word
possible_words = [word2 for (word1, word2) in bigrams if word1.lower() ==
generated_text[-1].lower()]

 # Choose a word randomly from the possible options
 next_word = random.choice(possible_words)
 generated_text.append(next_word)
# Print the generated text
print(' '.join(generated_text))
```

OUTPUT:

```
Requirement already satisfied: nltk in /usr/local/lib/python3.10/dist-packages (3.8.1)
Requirement already satisfied: click in /usr/local/lib/python3.10/dist-packages (from nltk)
(8.1.7)
Requirement already satisfied: joblib in /usr/local/lib/python3.10/dist-packages (from nltk)
(1.3.2)
Requirement already satisfied: regex>=2021.8.3 in /usr/local/lib/python3.10/dist-packages
(from nltk) (2023.6.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from nltk)
(4.66.1)
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data] Package punkt is already up-to-date!
[nltk_data] Downloading package gutenberg to /root/nltk_data...
[nltk_data] Package gutenberg is already up-to-date!
the sea , avoided . Hence it my life in the left to keep my witness shall no hesitation , accept
```

LAB 3: TEXT CLASSIFICATION

AIM: To perform Text classification using python and scikit-learn

ALGORITHM:

Algorithm: Text Classification using LinearSVC

1. Load the 20 Newsgroups dataset with specified categories.

 - Import the necessary libraries: fetch_20newsgroups from sklearn.datasets.

 - Specify the categories of interest for classification.

 - Use fetch_20newsgroups to load the dataset for both training and testing sets.

2. Split the dataset into training and testing sets.

 - Import train_test_split from sklearn.model_selection.

 - Split the dataset into X_train, X_test, y_train, and y_test.

3. Create a pipeline for text classification.

 - Import make_pipeline from sklearn.pipeline.

 - Create a pipeline with TF-IDF Vectorizer and LinearSVC classifier.

4. Train the model on the training data.

 - Call the fit method on the pipeline with X_train and y_train as input.

5. Predict labels for the testing data.

 - Use the trained model to predict labels for X_test.

6. Evaluate the model's performance.

 - Calculate accuracy_score to measure the accuracy of the model.

 - Print classification_report to see precision, recall, and F1-score for each class.

End Algorithm

PROGRAM:

```
# Install scikit-learn if not already installed
!pip install scikit-learn
# Import necessary libraries
import pandas as pd
from sklearn.datasets import fetch_20newsgroups
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.pipeline import make_pipeline
from sklearn.svm import LinearSVC
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
# Load the 20 Newsgroups dataset
categories = ['sci.med', 'sci.space', 'comp.graphics', 'talk.politics.mideast']
newsgroups_train = fetch_20 newsgroups(subset='train', categories=categories)
```

```
newsgroups_test = fetch_20newsgroups(subset='test', categories=categories)
# Split the data into training and testing sets
X_train = newsgroups_train.data
X_test = newsgroups_test.data
y_train = newsgroups_train.target
y_test = newsgroups_test.target
# Create a pipeline with TF-IDF vectorizer and LinearSVC classifier
model = make_pipeline(  TfidfVectorizer(), LinearSVC() )
# Train the model
model.fit(X_train, y_train)
# Predict labels for the test set
predictions = model.predict(X_test)
# Evaluate the model
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)
print("\nClassification Report:")
print(classification_report(y_test, predictions))
```

OUTPUT:

```
Requirement already satisfied: scikit-learn in
Accuracy: 0.9504823151125402
Classification Report:
 precision recall f1-score support
 0 0.89 0.97 0.93 389
 1 0.96 0.91 0.94 396
 2 0.98 0.94 0.96 394
 3 0.98 0.98 0.98 376
 accuracy 0.95 1555
 macro avg 0.95 0.95 0.95 1555
weighted avg 0.95 0.95 0.95 1555
```

LAB 4: SEMANTIC ANALYSIS

AIM: To perform Semantic Analysis using Gensim

ALGORITHM:

```
1. Install Necessary Libraries: Install Gensim and NLTK libraries (!pip install gensim,
!pip install nltk).
2. Import Libraries: Import required libraries: gensim for word vectors and nltk for
tokenization.
3. Download Pre-trained Word Vectors: Download pre-trained word vectors (Word2Vec)
using Gensim's api.load() method.
4. Define Sample Sentences: Create sample sentences for semantic analysis.
5. Tokenization: Tokenize the sentences into words using NLTK's word_tokenize()
method.
6. Semantic Analysis with Word Vectors: Iterate through each tokenized sentence.
For each word in the sentence:
Check if the word exists in the pre-trained Word2Vec model.
If the word exists:
Find words similar to the current word using word_vectors.most_similar(word).
Display or store the similar words.
If the word doesn't exist in the model:
Print a message indicating that the word is not in the pre-trained model.
```

PROGRAM:

```
# Install necessary libraries
!pip install gensim
!pip install nltk
# Import required libraries
import gensim.downloader as api
from nltk.tokenize import word_tokenize
# Download pre-trained word vectors (Word2Vec)
word_vectors = api.load("word2vec-google-news-300")
# Sample sentences
sentences = [
"Natural language processing is a challenging but fascinating field.",
"Word embeddings capture semantic meanings of words in a vector space."
]
# Tokenize sentences
tokenized_sentences = [word_tokenize(sentence.lower()) for sentence in sentences]
```

```
# Perform semantic analysis using pre-trained word vectors
for tokenized_sentence in tokenized_sentences:
    for word in tokenized_sentence:
        if word in word_vectors:
            similar_words = word_vectors.most_similar(word)
            print(f"Words similar to '{word}': {similar_words}")
    else:
            print(f"'{word}' is not in the pre-trained Word2Vec model.")
```

OUTPUT:

```
Requirement already satisfied: gensim in /usr/local/lib/python3.10/dist-packages (4.3.2)
Words similar to 'natural': [('Splittorff_lacked', 0.636509358882904), ('Natural',
0.58078932762146), ('Mike_Taugher_covers', 0.577259361743927), ('manmade',
0.5276211500167847), ('shell_salted_pistachios', 0.5084421634674072), ('unnatural',
0.5030758380889893), ('naturally', 0.49992606043815613), ('Intraparty_squabbles',
0.4988228678703308), ('Burt_Bees_®', 0.49539363384246826), ('causes_Buxeda',
0.4935200810432434)]
Words similar to 'language': [('langauge', 0.7476695775985718), ('Language',
0.6695356369018555), ('languages', 0.6341332197189331), ('English',
0.6120712757110596), ('CMPB_Spanish', 0.6083104610443115), ('nonnative_speakers',
0.6063109636306763), ('idiomatic_expressions', 0.5889801979064941), ('verb_tenses',
0.58415687084198), ('Kumeyaay_Diegueno', 0.5798824429512024), ('dialect',
0.5724600553512573)]
```

LAB 5: SENTIMENT ANALYSIS

AIM: To perform sentiment analysis program using an SVM classifier with TF-IDF vectorization

ALGORITHM:

1. Library Installation and Import: Install required libraries (scikit-learn and nltk). Import necessary modules from these libraries.

2. Download NLTK Resources: Download the movie_reviews dataset from NLTK.

3. Load and Prepare Dataset: Load the movie_reviews dataset. Convert the dataset into a suitable format (list of words and corresponding sentiments) and create a DataFrame.

4. Split Data into Train and Test Sets: Split the dataset into training and testing sets (e.g., 80% training, 20% testing).

5. TF-IDF Vectorization: Initialize a TF-IDF vectorizer. Fit and transform the training text data to convert it into numerical TF-IDF vectors.

6. Initialize and Train SVM Classifier: Initialize an SVM classifier (using a linear kernel for this example). Train the SVM classifier using the TF-IDF vectors and corresponding sentiment labels.

7. Prediction and Evaluation: Transform the test text data into TF-IDF vectors using the trained vectorizer. Predict sentiment labels for the test data using the trained SVM classifier. Calculate the accuracy score to evaluate the model's performance. Generate a classification report showing precision, recall, and F1-score for each class.

PROGRAM:

```
# Install necessary libraries

!pip install scikit-learn

!pip install nltk

# Import required libraries

import pandas as pd

from sklearn.model_selection import train_test_split

from sklearn.feature_extraction.text import TfidfVectorizer

from sklearn.svm import SVC

from sklearn.metrics import accuracy_score, classification_report

from nltk.corpus import movie_reviews # Sample dataset from NLTK

# Download NLTK resources (run only once if not downloaded)

import nltk

nltk.download('movie_reviews')

# Load the movie_reviews dataset

documents = [(list(movie_reviews.words(fileid)), category)

 for category in movie_reviews.categories()

 for fileid in movie_reviews.fileids(category)]

# Convert data to DataFrame

df = pd.DataFrame(documents, columns=['text', 'sentiment'])

# Split data into train and test sets

X_train, X_test, y_train, y_test = train_test_split(df['text'], df['sentiment'], test_size=0.2,
random_state=42)
# Initialize TF-IDF vectorizer
tfidf_vectorizer = TfidfVectorizer()
# Fit and transform the training data
X_train_tfidf = tfidf_vectorizer.fit_transform(X_train.apply(' '.join))
# Initialize SVM classifier
svm_classifier = SVC(kernel='linear')
# Train the classifier
svm_classifier.fit(X_train_tfidf, y_train)
# Transform the test data
X_test_tfidf = tfidf_vectorizer.transform(X_test.apply(' '.join))
# Predict on the test data
y_pred = svm_classifier.predict(X_test_tfidf)
# Calculate accuracy
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy:.2f}')
# Display classification report
print(classification_report(y_test, y_pred))
```

OUTPUT:

```
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.10/dist-packages (1.2.2)
[nltk_data] Downloading package movie_reviews to /root/nltk_data...
[nltk_data] Unzipping corpora/movie_reviews.zip.
Accuracy: 0.84
```

```
               Precision      recall     f1-score     support
 Neg            0.83          0.85       0.84          199
 Pos            0.85          0.82       0.84          201
 Accuracy                                0.84         400
 macro avg       0.84         0.84        0.84        400
weighted avg     0.84         0.84        0.84        400
```

LAB 6: PARTS OF SPEECH TAGGING

AIM: To perform Parts of Speech (POS) tagging program using NLTK

ALGORITHM:

1. Library Installation and Import: Install NLTK library if not already installed. Import the necessary NLTK library for text processing and POS tagging.

2. Download NLTK Resources: Download NLTK resources required for tokenization and POS tagging (punkt for tokenization, averaged_perceptron_tagger for POS tagging).

3. Sample Text: Define a sample text for POS tagging.

4. Tokenization: Break down the provided text into individual words (tokens) using NLTK's word_tokenize() method.

5. POS Tagging: Perform POS tagging on the tokens obtained from the text using NLTK's pos_tag() method. Assign POS tags to each word in the text based on its grammatical category (noun, verb, adjective, etc.).

6. Display POS Tags: Print or display the words along with their respective POS tags generated by the POS tagging process.

PROGRAM:

```
# Install NLTK (if not already installed)

!pip install nltk

# Import necessary libraries

import nltk

nltk.download('punkt')

nltk.download('averaged_perceptron_tagger')

# Sample text for POS tagging

text = "Parts of speech tagging helps to understand the function of each word in a sentence."

# Tokenize the text into words

tokens = nltk.word_tokenize(text)

# Perform POS tagging

pos_tags = nltk.pos_tag(tokens)

# Display the POS tags

print("POS tags:", pos_tags)
```

OUTPUT:

```
[nltk_data] /root/nltk_data...

[nltk_data] Unzipping taggers/averaged_perceptron_tagger.zip.

POS tags: [('Parts', 'NNS'), ('of', 'IN'), ('speech', 'NN'), ('tagging', 'VBG'), ('helps', 'NNS'), ('to', 'TO'), ('understand', 'VB'), ('the', 'DT'), ('function', 'NN'), ('of', 'IN'), ('each', 'DT'), ('word', 'NN'), ('in', 'IN'), ('a', 'DT'), ('sentence', 'NN'), ('.', '.')]
```

LAB 7: CHUNKING

AIM: To perform Noun Phrase chunking

ALGORITHM:

1. Import Necessary Libraries: Import required modules from NLTK for tokenization, POS tagging, and chunking.

2. Download NLTK Resources (if needed): Ensure NLTK resources like tokenizers and POS taggers are downloaded (nltk.download('punkt'), nltk.download('averaged_perceptron_tagger')).

3. Define a Sample Sentence: Set a sample sentence that will be used for chunking.

4. Tokenization: Break the sentence into individual words or tokens using NLTK's word_tokenize() function.

5. Part-of-Speech (POS) Tagging: Tag each token with its corresponding part-of-speech using NLTK's pos_tag() function.

6. Chunk Grammar Definition: Define a chunk grammar using regular expressions to identify noun phrases (NP). For example, NP: {<DT>?<JJ>*<NN>} captures sequences with optional determiners (DT), adjectives (JJ), and nouns (NN) as noun phrases.

7. Chunk Parser Creation: Create a chunk parser using RegexpParser() and provide the defined chunk grammar.

8. Chunking: Parse the tagged sentence using the created chunk parser to extract chunks based on the defined grammar.

9. Display Chunks: Iterate through the parsed chunks and print the subtrees labeled as 'NP', which represent the identified noun phrases.

PROGRAM:

```
!pip install nltk

import nltk

from nltk import RegexpParser

from nltk.tokenize import word_tokenize

from nltk.tag import pos_tag

# Download NLTK resources (run only once if not downloaded)

nltk.download('punkt')

nltk.download('averaged_perceptron_tagger')

# Sample sentence

sentence = "The quick brown fox jumps over the lazy dog"

# Tokenize the sentence

tokens = word_tokenize(sentence)

# POS tagging

tagged = pos_tag(tokens)

# Define a chunk grammar using regular expressions

# NP (noun phrase) chunking: "NP: {<DT>?<JJ>*<NN>}"

# This grammar captures optional determiner (DT), adjectives (JJ), and nouns (NN) as a noun

phrase

chunk_grammar = r"""  NP: {<DT>?<JJ>*<NN>} """
```

```
# Create a chunk parser with the defined grammar
chunk_parser = RegexpParser(chunk_grammar)
# Parse the tagged sentence to extract chunks
chunks = chunk_parser.parse(tagged)
# Display the chunks
for subtree in chunks.subtrees():
 if subtree.label() == 'NP': # Print only noun phrases
 print(subtree)
```

OUTPUT:

```
Requirement already satisfied: nltk in /usr/local/lib/python3.10/dist-packages (3.8.1)
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packages (from nltk)
(4.66.1)
(NP The/DT quick/JJ brown/NN)
(NP fox/NN)
(NP the/DT lazy/JJ dog/NN)
[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!
```