



SASTRA
ENGINEERING · MANAGEMENT · LAW · SCIENCES · HUMANITIES · EDUCATION
DEEMED TO BE UNIVERSITY
(U/S 3 of the UGC Act, 1956)



THINK MERIT | THINK TRANSPARENCY | THINK SASTRA

NATURAL LANGUAGE PROCESSING

Course Code: CSE405R02

Semester: V

Lab Manual

2025

SHANMUGHA ARTS, SCIENCE, TECHNOLOGY AND RESEARCH ACADEMY
(SASTRA Deemed to be) University
Tirumalaisamudram, Thanjavur – 613401
School of Computing

Course Objective:

This course will help the learner to describe the phases of Natural Language Processing (NLP) and to apply the concepts in specific applications.

Course Learning Outcomes:

Upon successful completion of this course, the learner will be able to

CO No.	COURSE OUTCOME	Knowledge Level
1	Recall the fundamental mathematical models in the field of Natural Language Processing	K1
2	Solve the collocation and word sense disambiguation problem using Natural Language Processing Techniques	K3
3	Construct Neural Language Models for solving classification problem	K3
4	Demonstrate different Natural Language Processing task using Recurrent Neural Networks	K2
5	Explain machine learning techniques used in NLP, such as hidden Markov models and probabilistic context-free grammars	K2
6	Illustrate the Machine Translation and Information Extraction using machine learning algorithms	K2

LIST OF EXERCISES

1. Perform the following task using NLTK: Tokenize and tag some text, identify named entities, display a parse tree and find the ambiguity of the sentence using parse tree.
2. Perform t-Test and Chi-Square test to check whether a given sequence of words is a collocation or not.
3. Implement the decision rule-based Naïve Bayes disambiguation method to find the sense of an ambiguous word with the given training set.
4. Implement the Hindle and Rooth algorithm for solving the attachment ambiguity problem.
5. Implement the forward and backward procedures using Hidden Markov Model to find the probability of a word sequence.
6. Implement the Viterbi algorithm to find the probability of a word sequence, and infer the best tag sequence using Hidden Markov Model.
7. Implement the Probabilistic Context Free Grammar (PCFG) and find the inside probability of a word sequence using the CYK algorithm.
8. Implement text classification using Bag of Words and TF-IDF with TensorFlow.
9. Implement word2vec model to explore the semantic similarity between the words.
10. Design text generator using LSTMs and generate words based on the input provided by the users with available corpus.
11. Create an RNN based Python machine translation system.
12. Develop a chatbot using Large Language Model.

LIBRARIES TO BE INSTALLED

Libraries	Description
NumPy	Numerical Computing Tool (supports large, multidimensional array and matrices functions)
Pandas	Data Analysis and Manipulation Tool
NLTK	Natural Language Toolkit for text processing and analysis
Keras	Python interface for Artificial Neural Networks
TensorFlow	Open-source framework for Machine Learning and Deep Learning, supports numerical computations on CPUs, GPUs, and cluster of GPUs.
Scikit-Learn	Open-source Data Analysis Library

PACKAGES TO BE INSTALLED

Package	Description
Punkt	Sentence tokenizer for multilingual text segmentation and tokenization tasks.
Maxent Treebank Pos Tagger	Part-of-speech tagger based on Maximum Entropy modelling for accurate linguistic analysis.
Stopwords	Contains common words removed during text analysis to focus on meaningful content
Wordnet	Lexical database of English, organized by synsets to explore word relationships and meanings.
Averaged Perceptron Tagger	POS tagger using the averaged perceptron algorithm for efficient and accurate tagging.
OMW	Open Multilingual WordNet, a lexical database linking words across languages for semantic analysis.

EXPERIMENT NO. – 1

Aim:

To perform the following task using NLTK: Tokenize and tag some text, identify named entities, display a parse tree and find the ambiguity of the sentence using parse tree.

Tokenizer, Stop words and Wordnet

Tokenizing is the process of breaking up a piece of text into smaller parts, such as sentences and words. These smaller parts are called tokens. E.g., a word is a token in a sentence, and a sentence is a token in a paragraph.

Stop words are those words that are present in text but don't contribute in the meaning of a sentence. Such words are not at all important for the purpose of information retrieval or natural language processing. E.g., 'the' and 'a'.

Wordnet is a large lexical database of English, created by Princeton. It is a part of NLTK corpus.

Stemming, Lemmatization, Word Replacement, Synonym and Antonym replacement

Stemming is a technique used to extract the base form of the words by removing affixes from them. E.g., the stem of the words eating, eats, eaten is eat.

Lemmatization technique is like stemming. The output after lemmatization is called 'lemma', which is a root word rather than root stem, the output of stemming. After lemmatization, we will be getting a valid word that means the same thing.

Word replacement can be thought of as text normalization or error correction.

Parse Tree

Parsing refers to the process of analysing the strings of symbols in natural language conforming to the rules of formal grammar. Parser is used to report any syntax error. It helps to recover from commonly occurring error so that the processing of the remainder of program can be continued. Parse tree is created with the help of parser. It is used to create symbol table, which plays an important role in NLP. It is also used to produce intermediate representations (IR).

Named Entity Recognition

Named Entity Recognition is a crucial task in natural language processing that involves identifying and classifying key elements in text into predefined categories such as person names, organizations, locations, dates, and more. It helps in structuring unstructured data, enabling better information retrieval, text summarization, and question-answering systems. It uses techniques like machine learning, rule-based methods, and deep learning to accurately detect and label entities, significantly enhancing the ability to understand and utilize textual data.

Procedure:

1. Install NLTK and download its dataset and packages.
2. Tokenize the given text into words and sentences using tokenizer module like *PunktSentenceTokenizer*.
3. Tag each word with its part of speech using *pos_tag* module of NLTK.
4. Identify named entities in the tagged text using *ne_chunk* module of NLTK.

5. Create and display a parse tree using a grammar and a parser.
6. Identify potential ambiguities using different parse trees.

Packages:

- *nltk.tokenize*: It is the package provided by NLTK module to achieve the process of tokenization.
- *nltk.corpus*: It defines a collection of corpus reader classes, which can be used to access the contents of a diverse set of corpora.
- *nltk.word_tokenize*: It is used for basic word tokenization.
- *PunktSentenceTokenizer*: It trains on raw text to produce a custom sentence tokenizer. Get raw text either by reading in a file or from an NLTK corpus using the *raw()* method.
- *nltk.pos_tag*: Part of speech tagging on a list of tokenized words using the *pos_tag* function from the NLTK library. Part-of-speech tagging involves grammatical categories (like nouns, verbs, adjectives, etc.) to each word in a sentence.
- *nltk.ne_chunk*: It is used to identify named entities in a tagged sentence (where each word has been assigned a part-of-speech tag).
- *en_core_web_sm*: It is a small pre-trained model for English text processing. It includes capabilities like part-of-speech tagging, named entity recognition, dependency parsing, etc.

Various stemmers in stem module:

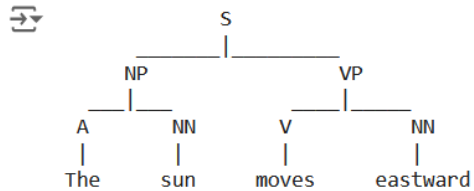
- *PorterStemmer* class: It has several regular word forms and suffixes with the help of which it can transform input word to a final stem.
- *LancasterStemmer* class: NLTK has LancasterStemmer class with the help of which we can easily implement Lancaster Stemmer algorithms for the word we want to stem
- *RegexpStemmer* class: It takes a single regular expression and removes any prefix or suffix that matches the expression.
- *SnowballStemmer* class: It supports 15 non-English languages. It creates an instance with the specified language and then call *stem()* method.

Sample input and output:

Parse Tree

```
⇒ (S (NP (A The) (NN sun)) (VP (V moves) (NN eastward))))
```

```
[12] 1 tree.pretty_print()
```



Result:

Successfully tokenized and tagged the text, identified named entities, displayed parse trees, and analysed the sentence ambiguity using NLTK.

EXPERIMENT NO. – 2

Aim:

To perform t-Test and Chi-Square test to check whether a given sequence of words is a collocation or not.

Collocation

Collocation refers to the habitual juxtaposition of a particular word with another word or words with a frequency greater than chance. In linguistics, collocations are combinations of words that are often found together in a language, providing natural and fluent expressions. E.g., make a decision, strong tea, heavy rain, take a break.

Importance of collocation

1. Natural Language Processing: Collocations help in various NLP tasks such as machine translation, information retrieval, and text generation by ensuring that the output is natural and coherent.
2. Language Learning: For language learners, understanding collocations is crucial for achieving fluency and sounding more like a native speaker.
3. Lexicography: In dictionary compilation, collocations provide essential information about how words are used in context.

Identifying collocations

Statistical measures are commonly used to identify collocations in a corpus. Two popular statistical tests are:

1. *t-Test*: It is used to determine if there is a significant difference between the expected and observed frequencies of word pairs.
2. *Chi-Square Test*: It is used to test the independence of words, checking if the occurrence of one word is independent of the occurrence of another.

t-Test

A t-Test is a statistical test used to compare the means of two groups to determine if they are significantly different from each other. It is widely used in hypothesis testing to assess whether the means of two groups are equal. The assumptions are that the data is continuous, approximately normally distributed, with equal variances between two groups.

$$t = \frac{\bar{x} - \mu}{\sqrt{\frac{s^2}{N}}}$$

where,

$\bar{x}$ is the sample mean

μ is the expected mean of the distribution

s^2 is the sample variance

N is the sample size

Chi-Square Test

The Chi-Square Test is a non-parametric statistical test used to determine if there is a significant association between two categorical variables. It compares the observed frequencies of events

with the expected frequencies under the assumption of independence. The assumptions are that the data is categorical, and the samples are independent. It is applied to a 2x2 table of observed frequencies.

$$\chi^2 = \sum_{i,j} \frac{(O_{i,j} - E_{i,j})^2}{E_{i,j}}$$

where,

i ranges over rows

j ranges over columns

$O_{i,j}$ is the observed value for cell (i,j)

$E_{i,j}$ is the expected value for cell (i,j)

Procedure:

I. t-Test

- Import necessary packages like – *nltk*, *pandas*, etc.
- Collect a large corpus of text data that includes the sequence of words that we want to test for collocation.
- Split the text into words or tokens after removing punctuations and stop words.
- Count the frequency of adjacent words with part of speech filters.
- Calculate the Pointwise Mutual Information (PMI) for bigrams and trigrams with more than one occurrence and sorting them by PMI in descending order.
- Calculate and sort the t-scores for bigrams and trigrams, then filter them to retain only those with specific part-of-speech patterns.

II. Chi-square Test

- Import necessary packages like – *nltk*.
- Collect a large corpus of text data that includes the sequence of words that we want to test for collocation.
- Split the text into words or tokens after removing punctuations and stop words.
- Implement a Chi-Square test to determine if two given words form a collocation and return the result if they do.

Sample input and output:

Input: Based on any given dataset

 t-test:

```
t-value: 1.1      Words: knocked and door are not collocated!
t-value: 0.64     Words: door and knocked are not collocated!
t-value: 1.1      Words: knocked and door are not collocated!
t-value: 0.73     Words: door and man are not collocated!
t-value: 0.64     Words: man and knocked are not collocated!
t-value: 0.64     Words: knocked and metal are not collocated!
t-value: 0.91     Words: metal and front are not collocated!
t-value: 0.73     Words: front and door are not collocated!
t-value: 0.64     Words: door and knocked are not collocated!
t-value: 1.1      Words: knocked and door are not collocated!
```



chi^2-test:

chi-value: 7.218750000000001	Words: knocked and door are collocated!
chi-value: 1.6369047619047623	Words: door and knocked are not collocated!
chi-value: 7.218750000000001	Words: knocked and door are collocated!
chi-value: 2.933333333333333	Words: door and man are collocated!
chi-value: 1.925	Words: man and knocked are not collocated!
chi-value: 1.925	Words: knocked and metal are not collocated!
chi-value: 10.999999999999996	Words: metal and front are collocated!
chi-value: 2.933333333333336	Words: front and door are collocated!
chi-value: 1.6369047619047623	Words: door and knocked are not collocated!
chi-value: 7.218750000000001	Words: knocked and door are collocated!

Result:

The t-Test and Chi-Square test successfully identified significant collocations within the given sequence of words.

EXPERIMENT NO. – 3

Aim:

To implement the decision rule-based Naïve Bayes disambiguation method to find the sense of an ambiguous word with the given training set.

Naïve Bayes Disambiguation

The process of determining the correct meaning of a word in context when the word has multiple meanings (senses). The Naïve Bayes Disambiguation method is a probabilistic approach used in natural language processing to resolve ambiguities in word meaning, known as word sense disambiguation (WSD). This method applies the Naïve Bayes algorithm, which is based on Bayes' Theorem with the assumption of independence among features.

Naïve Bayes is widely used in ML due to its ability to efficiently combine evidence from a wide variety of features. It can be applied if the state of the world we base our classification on can be described as a set of attributes. It assumes that all features (context words) are conditionally independent given the sense, which simplifies the computation but may not always hold true in practice. Although the Naïve Bayes assumption is incorrect in the context of text processing, it often does quite well, partly because the decisions made can be optimal even in the face of the inaccurate assumption.

Bayesian Disambiguation Algorithm

Training

```
for all senses  $s_k$  of  $w$  do
    for all  $v_j$  in vocabulary do
         $P(v_j|s_k) = C(v_j, s_k)/C(s_k)$ 
    end
end
for all senses  $s_k$  of  $w$  do
     $P(s_k) = C(s_k)/C(w)$ 
end
```

Disambiguation

```
for all senses  $s_k$  of  $w$  do
     $score(s_k) = \log P(s_k)$ 
    for all  $v_j$  in context window  $c$  do
         $score(s_k) = score(s_k) + \log P(v_j|s_k)$ 
    end
end
choose  $\arg \max s_k score(s_k)$ 
```

Sample input and output:

```
print("The sense of 'Chair' is furniture.", final_score_f)
else:
    print("The sense of 'Chair' is position.", final_score_p)
```



The sense of 'Chair' is furniture. -4.221848749616356

Result:

The decision rule-based Naïve Bayes disambiguation method successfully identified the most likely sense of the ambiguous word using the given training set.

EXERCISE NO. – 4

Aim:

To implement the Hindle and Rooth algorithm for solving the attachment ambiguity problem.

Attachment Ambiguity

Attachment Ambiguity occurs in natural language processing when a phrase or clause can be associated with (or “attached” to) different part of a sentence, leading to multiple possible interpretations. This type of ambiguity is common in complex sentences where prepositional phrases, relative clauses, or adverbial phrases can attach to different words or phrases within the sentence. E.g., John saw the man [with the telescope].” (The man has the telescope.) and John [saw the man] with the telescope.” (John used the telescope to see the man.)

Hindle and Rooth algorithm

The algorithm is a statistical approach to resolving attachment ambiguity in natural language processing specifically focusing on prepositional phrase (PP) attachment. The algorithm leverages lexical statistics derived from large corpora to determine the most likely attachment site for ambiguous prepositional phrase.

The algorithm defines the event space to include all clauses containing a transitive verb, an NP following the verb, and a PP following the NP. To reduce the complexity of the model, one preposition is attended at a time. In cases where two PPs with the same preposition appear in sequence, it models only the behaviour of first preposition. To simplify it, instead of directly asking whether a specific preposition is attached to a particular verb or noun, we estimate the general likelihood of a preposition attaching to a verb or a noun. This approach allows it to make informed decisions about PP attachment based on statistical probabilities derived from a large corpus, thus enabling resolution of PP attachment ambiguity in language processing.

$$\begin{aligned}\lambda(v, n, p) &= \log_2 \frac{P(\text{Attach}(p) = v|v, n)}{P(\text{Attach}(p) = n|v, n)} \\ &= \log_2 \frac{P(VA_p = 1|v)P(NA_p = 0|v)}{P(NA_p = 1|n)}\end{aligned}$$

where,

$$P(VA_p = 1|v) = \frac{C(v, p)}{C(v)}$$

$$P(NA_p = 1|n) = \frac{C(n, p)}{C(n)}$$

Procedure:

1. Collect a large corpus of text that contains sentences with prepositional phrases (PPs).
2. Use a syntactic parser to parse sentences in the corpus, identifying verbs, nouns, and PPs.
3. Extract clauses that contain a transitive verb, an NP (object noun phrase) following the verb, and a PP following the NP.
4. Set up counters to keep track of the frequency of verb-preposition-noun (V-P-N) and noun-preposition-noun (N-P-N) combinations.

5. Iterate through the extracted clauses and count occurrences of V-P-N and N-P-N combinations.
6. Computer prior probability for each attachment type (verb attachment and noun attachment) by dividing the frequency of each type by the total number of occurrences.
7. Estimate conditional probabilities for each V-P-N and N-P-N combination.
8. For a new sentence containing a potentially ambiguous PP, parse the sentence to identify the verb, noun, and preposition.
9. Use the probabilities computed earlier to calculate the likelihood of the PP attaching to the verb versus the noun.
10. Compare the probabilities and select the attachment (verb or noun) with the higher probability as the correct interpretation.

Sample input and output:

Enter a word: Ball

```
print("Verb")
else:
    print("Noun")
```

Noun

Result:

Hindle and Rooth algorithm has been successfully implemented for solving attachment ambiguity problem.

EXERCISE NO. – 5

Aim:

To implement the forward and backward procedures using Hidden Markov Model to find the probability of a word sequence.

Hidden Markov Model (HMM)

The Hidden Markov Model (HMM) is a statistical framework designed to represent the probabilistic connections between a series of observed events and a sequence of hidden states. This model is particularly useful in scenarios where the system or process responsible for producing the observations is not directly observable, thus the term ‘Hidden Markov Model’.

It is utilized for predicting future observations or classifying sequences based on the hidden processes generating the data. It is composed of 2 types of variables: hidden states and observations. Hidden states are the underlying factors that produce the observed data but cannot be directly observed. Observations are the measurable and observable variables.

The forward and backward procedures are essential algorithms in HMMs used to compare the probability of a sequence of observations. The forward procedure calculates the probability of observing a sequence of events up to a certain point, given a specific hidden state at that point. It works by recursively summing over all possible previous states and their transitions. The backward procedure calculates the probability of the ending part of the observation sequence, given a specific hidden state at the current point. It works by recursively summing over all possible subsequent states and their transitions.

Forward Algorithm

1. Initialization

$$\alpha_1(j) = \pi_j b_j(o_1)$$

2. Recursion

$$\alpha_t(j) = \sum_{i=1}^N \alpha_{t-1}(i) \cdot a_{ij} \cdot b_j(o_t)$$

3. Termination

$$P(O|\lambda) = \sum_{j=1}^N \alpha_T(j)$$

Backward Algorithm

1. Initialization

$$\beta_T(j) = 1$$

2. Recursion

$$\beta_t(i) = \sum_{j=1}^N \beta_{t+1}(j) \cdot a_{ij} \cdot b_j(o_{t+1})$$

3. Termination

$$P(O|\lambda) = \sum_{j=1}^N \pi_j \cdot b_j(o_1) \cdot \beta_1(j)$$

Sample input and output:

Input:

Enter no of states: 2
Initial probability for state: 1.0
State Name: R
Initial probability for state: 0
State Name: S
{0: 1.0, 1: 0.0}

Output:

Emission States: A B
Output Emissions: A A B
Possible Emission: 4
Emission Probability for sequence i to j: 0.8
Emission Probability for sequence i to j: 0.2
Emission Probability for sequence i to j: 0.3
Emission Probability for sequence i to j: 0.7
{(0, 'A'): 0.8, (0, 'B'): 0.2, (1, 'A'): 0.3, (1, 'B'): 0.7}

Gamma values: [[1.0, 0.0], [0.6956521739130435, 0.3043478260869566],
[0.20869565217391303, 0.7913043478260869], [0.12521739130434784,
0.8747826086956522]]
4
State0: [1.0, 0.6956521739130435, 0, 0]
State1: [0, 0, 0.7913043478260869, 0.8747826086956522]
State2: []

Result:

The forward and backward procedures using the Hidden Markov Model successfully calculated the probability of the given word sequence.

EXERCISE NO. – 6

Aim:

To implement the Viterbi algorithm to find the probability of a word sequence, and infer the best tag sequence using Hidden Markov Model.

Viterbi algorithm

The Viterbi algorithm is a fundamental dynamic programming technique widely used in the context of Hidden Markov Models (HMMs) to uncover the most likely sequence of hidden states given a sequence of observed events.

Given a series of observed events, the Viterbi algorithm determines the most likely order of hidden states in an HMM. The most likely sequence of states is computed efficiently by the Viterbi method, which divides the issue into smaller subproblems and solves them recursively.

It operates by iteratively computing the highest probability paths to each state at each state at each time step, storing these probabilities, and backtracking to determine the most probable sequence of hidden states. This method ensures an efficient and accurate decoding of hidden state sequences. Using dynamic programming, we calculate the most probable path as follows:

1. Initialization

$$\delta_j(1) = \pi_j, \quad 1 \leq j \leq N$$

2. Induction

$$\delta_j(t+1) = \max_{1 \leq i \leq N} \delta_i(t) a_{ij} b_{ij} O_t \quad 1 \leq j \leq N$$

Store backtrack

$$\psi_j(t+1) = \arg \max_{1 \leq i \leq N} \delta_i(t) a_{ij} b_{ij} O_t \quad 1 \leq j \leq N$$

3. Termination and path readout (by backtracking). The most likely state sequence is worked out from the right backwards.

$$\hat{X}_{T+1} = \arg \max_{1 \leq i \leq N} \delta_i(T+1)$$

$$\hat{X}_t = \psi_{\hat{X}_{T+1}}(t+1)$$

$$P(\hat{X}) = \max_{1 \leq i \leq N} \delta_i(T+1)$$

Sample input and output:

```
1 states = ('Rainy', 'Sunny')
2 observations = ('Walk', 'Shop', 'Clean')
3 start_probability = {'Rainy': 0.6, 'Sunny': 0.4}
4 transition_probability = {
5     'Rainy' : {'Rainy': 0.7, 'Sunny': 0.3},
6     'Sunny' : {'Rainy': 0.4, 'Sunny': 0.6},
7 }
8 emission_probability = {
9     'Rainy' : {'Walk': 0.1, 'Shop': 0.4, 'Clean': 0.5},
10    'Sunny' : {'Walk': 0.6, 'Shop': 0.3, 'Clean': 0.1},
11 }
12
13 print(viterbi(observations, states, start_probability, transition_probability, emission_probability))
```

 (0.01344, ['Sunny', 'Rainy', 'Rainy'])

Result:

The Viterbi algorithm successfully determined the most likely sequence of tags and calculated the probability of the given word sequence using the Hidden Markov Model.

EXERCISE NO. – 7

Aim:

To implement the Probabilistic Context Free Grammar (PCFG) and find the inside probability of a word sequence using the CYK algorithm.

Cocke-Younger-Kasami (CYK) algorithm

This algorithm is a parsing algorithm for context-free grammars, particularly suited for determining whether a given string can be generated by a given grammar. It is a dynamic programming algorithm that is most effective for grammars in Chomsky Normal Form (CNF).

function CYK(*words, grammar*) **returns** The most probable parse and its probability

Create and clear $\pi[num_words, num_wors, num_nonterminals]$

base case

for $i \leftarrow 1$ **to** num_words

for $A \leftarrow 1$ **to** $num_nonterminals$

if $(A \rightarrow w_i)$ is in grammar **then**

$\pi[i, i, A] \leftarrow P(A \rightarrow w_i)$

recursive case

for $span \leftarrow 2$ **to** num_words

for $begin \leftarrow 1$ **to** $num_words - span + 1$

$end \leftarrow begin + span - 1$

for $m = begin$ **to** $end - 1$

for $A = 1$ **to** $num_nonterminals$

for $B = 1$ **to** $num_nonterminals$

for $C = 1$ **to** $num_nonterminals$

$prob = \pi[begin, m, B] \times \pi[m + 1, end, C] \times P(A \rightarrow$

$BC)$

if $(prob > \pi[begin, end, A])$ **then**

$\pi[begin, end, A] = prob$

$back[begin, end, A] = \{m, B, C\}$

return $build_tree(back[1, num_words, 1]), \pi[1, num_words, 1]$

Viterbi PCFG Algorithm

Viterbi PCFG is an extension of the Viterbi algorithm applied in the context of Probabilistic Context-Free Grammars (PCFG). PCFGs are a type of context-free grammar that associates probabilities with each of its production rules, enabling the calculation of the likelihood of different parse trees for a given sentence.

Steps involved:

1. Initialization

$$\delta_i(p, p) = P(N^i \rightarrow w_p)$$

2. Induction

$$\delta_i(p, q) = \max_{\substack{1 \leq j, k \leq n \\ p \leq r < q}} P(N^i \rightarrow N^j N^k) \delta_j(p, r) \delta_k(r + 1, q)$$

Store Backtrace

$$\psi_i(p, q) = \arg \max_{(j, k, r)} P(N^i \rightarrow N^j N^k) \delta_j(p, r) \delta_k(r + 1, q)$$

3. Termination and Path Readout

The probability of the most probable parse rooted in the start symbol is: $P(\hat{t}) = \delta_1(1, m)$.

The most probable tree $\hat{t}$ can be reconstructed as follows:

1. The root node of the most probable parse is N_{1m}^1 .
2. Let N_{pq}^i be the most probable parse and $\psi_i(p, q) = (j, k, r)$ the corresponding backtrace. Then the left and right daughters of N_{pq}^i are:

$$left(N_{pq}^i) = N_{pr}^j$$

$$right(N_{pq}^i) = N_{(r+1)q}^k$$

If there is no unique maximum in the computation of δ and ψ , then we simple choose one maximum at random.

Sample input and output:

Input: The cat chased the dog

```

↔ Inside probability of S: 21.400000000000002
   Inside probability of NP: 18.479999999999997
   Inside probability of VP: 17.120000000000005
   Inside probability of Det: 0.0
   Inside probability of N: 0.0

```

Result:

The CYK algorithm was successfully implemented to compute the inside probability of a word sequence using a Probabilistic Context-Free Grammar (PCFG).

EXERCISE NO. – 8

Aim:

To implement text classification using Bag of Words and TF-IDF with TensorFlow.

Text Classification

It is a fundamental task in NLP that involves assigning predefined categories to text documents. It has a wide range of applications, including sentiment analysis, spam detection, topic labelling, and more. The main goal is to develop models that can automatically classify text into different categories based on its content.

Bag of Words (BoW)

It is a simple and widely used model in NLP for representing text data. In BoW, a text document is represented as an unordered collection of words, disregarding grammar, and word order but keeping multiplicity. The main steps in creating a BoW representation are:

1. Tokenization: Splitting the text into individual words or tokens.
2. Vocabulary Building: Creating a set of all unique words (vocabulary) from the corpus.
3. Vectorization: Representing each document as a vector, where each element corresponds to the frequency of a word in the document.

Term Frequency-Inverse Document Frequency (TF-IDF)

TF-IDF is an improvement over the BoW model that considers the importance of words in a document relative to the entire corpus. It combines two metrics:

1. *Term Frequency (TF)*: Measures the frequency of a word in a document.

$$TF(t, d) = \frac{\text{Number of times term } t \text{ appears in document } d}{\text{Total number of terms in document } d}$$

2. *Inverse Document Frequency (IDF)*: Measures the importance of a word in the corpus.

$$IDF(t, D) = \log \left(\frac{\text{Total number of documents}}{\text{Number of documents containing term } t} \right)$$

The TF-IDF score of a term in a document is the product of its TF and IDF values. This model helps to reduce the weight of common words and increase the weight of rare and informative words.

TensorFlow

It is an open-source machine learning library developed by Google. It provides a comprehensive ecosystem for building and deploying machine learning models, including tools for text processing, vectorization, and neural network training. It supports various deep learning architectures, making it suitable for complex text classification tasks.

Procedure:

1. Import all the required libraries and modules like *numpy*, *pandas*, *matplotlib*, *tensorflow*, etc.
2. Choose a large corpus of text and perform data cleaning steps like punctuation removal, stop-word removal, lemmatization, etc.

3. Split the data into train and validation sets and create TensorFlow dataset objects for training and validation dataset.
4. Use the *TextVectorization* method provided by TensorFlow to implement Bag of Words and TF-IDF.
5. Build a separate model for each Bag of Words and TF-IDF, compile it, and finally train the model.

Sample input and output:

```

Bag of Words Vocabulary:
['ai' 'enjoy' 'future' 'learning' 'love' 'machine' 'powerful'
 'programming' 'tensorflow' 'tool']

TF-IDF Vocabulary:
['ai' 'enjoy' 'future' 'learning' 'love' 'machine' 'powerful'
 'programming' 'tensorflow' 'tool']

Bag of Words Representation (BoW Table):
  ai  enjoy  future  learning  love  machine  powerful  programming  \
0   1     0     1         0     0         0         0         0
1   0     1     0         1     0         1         0         0
2   0     0     0         0     1         0         0         1
3   0     0     0         0     0         0         1         0

      tensorflow  tool
0              0    0
1              0    0
2              0    0
3              1    1

TF-IDF Representation (TF-IDF Table):
      ai  enjoy  future  learning  love  machine  powerful  \
0  0.707107  0.00000  0.707107  0.00000  0.000000  0.00000  0.00000
...
2    0.707107    0.00000  0.00000
3    0.000000    0.57735  0.57735

```

Result:

The text classification model using Bag of Words and TF-IDF with TensorFlow was successfully implemented.

EXERCISE NO. – 9

Aim:

To implement word2vec model to explore the semantic similarity between the words.

Word Embeddings

Word vectors are mathematical descriptions of individual words such that words that appear frequently together in the language will have similar values. In this way we can mathematically derive context.

They are a type of word representation that allows words to be represented as vectors in a continuous vector space. These embeddings capture semantic meanings and relationships between words based on their usage in large corpora of text. Unlike traditional methods like Bag of Words (BoW) or TF-IDF, which treat words as independent identities, word embeddings can capture contextual similarity and differences between words.

Word2Vec Model

It is a popular model for generating word embeddings that capture the semantic relationships between words. By leveraging large corpora and training efficient neural network models, Word2Vec can produce vector representations that are useful for various NLP tasks.

It is designed to transform words into high-dimensional vectors that reflect their semantic similarity. The main intuition behind Word2Vec is that words appearing in similar contexts tend to have similar meanings.

It utilized 2 architectures:

1. *CBOW (Continuous Bag of Words)*: It predicts the current word based on the surrounding context words within a defined window. The context words are fed into the input layer, while the output layer predicts the current word. The hidden layer holds the dimensions used to represent the current word found in the output layer.
2. *Skip Gram*: It predicts the surrounding context words within a specified window based on the current word. The current word is provided at the input layer, while the context words are predicted at the output layer. The hidden layer represents the number of dimensions used to encode the current word from the input layer.

Semantic Similarity

It is a metric defined over a set of documents or terms, where the idea of distance between items is based on the likeness of their meaning or semantic content. It includes “is a” relations. E.g., “car” is similar to “bus”.

Once the Word2Vec model is trained, it can be used to explore semantic similarity between words. The key idea is that words with similar meanings will have embeddings that are close to each other in the vector space. This can be measured using the cosine similarity, which quantifies the cosine of the angle between two vectors. Words with higher cosine similarity are considered more semantically similar.

Procedure:

1. Import all the required libraries and modules like *gensim*, *nltk*, etc.
2. Choose a large corpus of text and perform data cleaning steps like punctuation removal, stop-word removal, lemmatization, etc.
3. Split the text into words or tokens after removing punctuations and stop words.
4. Create CBOW model and Skip Gram model.
5. Print the results.

Sample input and output:

```
Cosine similarity between 'alice' and 'wonderland' - CBOW : 0.999249298413
Cosine similarity between 'alice' and 'machines' - CBOW : 0.974911910445
Cosine similarity between 'alice' and 'wonderland' - Skip Gram : 0.885471373104
Cosine similarity between 'alice' and 'machines' - Skip Gram : 0.856892599521
```

Result:

The Word2Vec model was successfully implemented, effectively capturing and exploring the semantic similarity between words.

EXERCISE NO. – 10

Aim:

To design text generator using LSTMs and generate words based on the input provided by the users with available corpus.

Text Generation using Long Short-Term Memory (LSTM)

Text Generation is a type of Language Modelling problem. It is the core problem for a number of natural language processing tasks like speech to text, and text summarization. A trained language model learns the likelihood of occurrence of a word based on the previous sequence of words used in the text. Language models can be operated at character level, n-gram level, sentence level or even paragraph level.

In LSTM, gates decide what data to be ignored and what to be feed-forward for the training. There are 3 gates in LSTM –

1. *Forget Gate*: This gate selects relevant information and discards irrelevant data, passing the relevant information through the input gate. The current and previous hidden states are processed using a sigmoid activation function, which outputs values between 0 and 1; values near 0 indicate the information should be ignored, while values closer to 1 indicate it should be passed through the input gate.
2. *Input Gate*: This gate adds information to the model using the sigmoid activation function. It creates an array of values ranging from -1 to 1 with the tanh activation function. The sigmoid function then filters this array to determine which information should be added to the model and which should be discarded.
3. *Output Gate*: The output gate generates the next hidden states and cell states for the next time step, using the tanh activation function to create a hidden state with values ranging from -1 to 1.

Procedure:

1. Load the necessary libraries like libraries for data handling viz. *pandas*, *numpy*, and deep learning libraries viz. *tensorflow*, *keras*.
2. Choose a large corpus of text and perform data cleaning steps like punctuation removal, stop-word removal, lemmatization, etc.
3. Generate n-gram sequence for training.
4. Create LSTM model that will take *predictors* as input X and *labels* as input Y and train it.
5. Finally, generate the text from the trained model architecture.

Sample input and output:

Input: Key terms taken from dataset.

```

print (generate_text("united states", 10, model, max_sequence_len))
print (generate_text("president trump", 10, model, max_sequence_len))
print (generate_text("donald trump", 10, model, max_sequence_len))
print (generate_text("india and china", 10, model, max_sequence_len))
print (generate_text("new york", 10, model, max_sequence_len))
print (generate_text("science and technology", 10, model, max_sequence_len))

1/1 _____ 1s 863ms/step
1/1 _____ 0s 84ms/step
1/1 _____ 0s 88ms/step
1/1 _____ 0s 90ms/step
1/1 _____ 0s 87ms/step
1/1 _____ 0s 97ms/step
1/1 _____ 0s 223ms/step
1/1 _____ 0s 88ms/step
1/1 _____ 0s 96ms/step
1/1 _____ 0s 99ms/step
United States Talks Syria Policy On Caveats Than A Fake Is Coulter

```

Result:

The LSTM-based text generator was successfully designed, generating 30 words based on user input.

EXERCISE NO. – 11

Aim:

To design RNN-based python machine translation system.

Recurrent Neural Network (RNN)

An RNN is a type of neural network where the output from the previous step is used as input for the current step. Unlike traditional neural networks, where all inputs and outputs are independent, RNNs are designed to handle tasks like predicting the next word in a sentence, where remembering previous words is essential. This need for retaining past information led to the development of RNNs, which address this challenge using a hidden layer.

The key feature of RNNs is their hidden state, which retains information about a sequence and is also known as the memory state because it remembers previous inputs to the network. RNNs use the same parameters for each input since they perform the same task on all inputs or hidden layers to produce the output, reducing parameter complexity compared to other neural networks.

Procedure:

1. Import essential libraries like *numpy* for numerical operations, *tensorflow* for building and training the neural network.
2. Create or load a dataset containing pairs of source and target language sentences. It will be used for training the machine translation model.
3. Build a vocabulary of unique characters from both the source and target datasets. Create a dictionary to map each character to a unique integer for both source and target languages.
4. Using the dictionaries created, convert each sentence in the source and target datasets into a sequence of integers that represent the corresponding characters.
5. Ensure all sequences have the same length by padding them with special character. This is necessary for efficient processing in batches.
6. Design a sequence-to-sequence model using TensorFlow. Define the input shape, layers, and compile the model with appropriate loss functions and optimizers.

7. Convert the target integer sequences into one-hot encoded vectors. This step is important for training the model to predict the correct character at each position in the target sequence.
8. Feed the padded and one-hot encoded sequences into the model. Specify the number of epochs, batch size, and validation split. Train the model to learn the translation from the source language to the target language.
9. Prepare a new input sentence by converting it into an integer sequence and padding it to match the training data format.
10. Use the trained model to predict the target sequence from the input sequence.

Sample input and output:

```
ProjectGurukul Project: English to French Translation  
Input sentence: Go.  
Decoded sentence: Va !
```

```
ProjectGurukul Project: English to French Translation  
Input sentence: Run!  
Decoded sentence: Cours !
```

Result:

The RNN-based Python machine translation system was successfully designed to translate text between languages.

EXERCISE NO. – 12

Aim:

To develop a chatbot using Large Language Model (LLM).

Large Language Model (LLM)

Large Language Models are used to develop human like conversations. It is a part of Natural Language Processing which focuses on enabling computers to understand, interpret, and generate human language. With the help of LLM, chatbots will be able to generate more human like responses, providing the conversation in a natural way.

Large Language Models stand as formidable entities within the field of artificial intelligence, distinguished by their intricate neural network architectures. These models excel at processing and generating text in a manner that mirrors human understanding. Whether handling simple greetings or navigating through complex discussions, LLMs demonstrate a profound ability to comprehend and contextualize language nuances. Beyond mere programs, they function akin to evolving digital brains, specializing in sophisticated text analysis and generation tasks.

Procedure:

1. Gather a large dataset of conversational text as the foundational training data for the chatbot.
2. Clean and preprocess the dataset by removing noise, normalizing text, handling special characters, and tokenizing it into meaningful units.
3. Choose a suitable pre-trained Large Language Model such as GPT-3 or GPT-4 based on the chatbot's requirements.
4. Fine-tune the selected LLM on the domain-specific conversational dataset to adapt the model's weights and response style.
5. Set up the development environment using Python with necessary libraries like transformers, openai, and torch.
6. Implement the input pipeline to capture user queries, preprocess them, and format them as prompts for the model.
7. Use the LLM to generate context-aware responses based on the processed input using appropriate inference techniques or API calls.
8. Develop a simple frontend or interface (CLI/web) to facilitate human-chatbot interaction.
9. Integrate memory or session management to maintain conversation history and context over multiple turns.
10. Conduct evaluation and error handling by testing chatbot responses, logging interactions, and iteratively improving prompts and model parameters.

Sample input and output:

```
[ ] 1 #Running chain through LLM with query
    2 query = "what types of perfumes do you sell?"
    3 response = chain.run(query)
    4 print_response(response)
```

Result:

The chatbot developed using a Large Language Model successfully engages users with contextually relevant and coherent conversational responses.