Table of Contents

Experiment No: Group 1-1

Title: To Perform tokenization (Whitespace, Punctuation-based, Treebank, Tweet, MWE) using NLTK library.

Objective: To Execute tokenization using Use porter stemmer and snowball stemmer .

Problem Statement: Perform tokenization (Whitespace, Punctuation-based, Treebank, Tweet, MWE) using NLTK library. Use porter stemmer and snowball stemmer for stemming. Use any technique for lemmatization. Input / Dataset –use any sample sentence.

Theory:

Tokenization
What is Tokenization?
Tokenization is a process of converting raw data into a useful data string. Tokenization is used in NLP for splitting paragraphs and sentences into smaller chunks that can be more easily assigned meaning.
Tokenization can be done to either at word level or sentence level. If the text is split into words it is called word tokenization and the separation done for sentences is called sentence tokenization.

Why is Tokenization required?
In tokenization process unstructured data and natural language text is broken into chunks of information that can be understood by machine.

Challenges of Tokenization :

- Dealing with segment words when spaces or punctuation marks define the boundaries of the word. For example: donâ€™t
- Dealing with symbols that might change the meaning of the word significantly. For example: ₹100 vs 100
- Contractions such as ‘you’re’ and ‘I’m’ should be properly broken down into their respective parts. An improper tokenization of the sentence can lead to misunderstandings later in the NLP process.

Types of Tokenization

1. Word Tokenization

- Most common way of tokenization, uses natural breaks, like pauses in speech or spaces in text, and splits the data into its respective words using delimiters (characters like ‘,’ or ‘;’ or “‘,”’).
- Word tokenization’s accuracy is based on the vocabulary it is trained with. Unknown words or Out Of Vocabulary (OOV) words cannot be tokenized.

2. White Space Tokenization

- Simplest technique, Uses white spaces as basis of splitting.
- Works well for languages in which the white space breaks apart the sentence into meaningful words.

3. Rule Based Tokenization

- Uses a set of rules that are created for the specific problem.
- Rules are usually based on grammar for particular language or problem.

4. Regular Expression Tokenizer

- Type of Rule based tokenizer
- Uses regular expression to control the tokenization of text into tokens.

5. Penn Treebank Tokenizer

- Penn Treebank is a corpus maintained by the University of Pennsylvania containing over four million and eight hundred thousand annotated words in it, all corrected by humans
- Uses regular expressions to tokenize text as in Penn Treebank

Stemming :
Stemming is a process of reducing inflectional words to their root form. It maps the word to a same stem even if the stem is not a valid word in the language.

English language has several variants of a single term. The presence of these variances in a text corpus results in data redundancy when developing NLP or machine learning models. Such models may be ineffective.

Types of Stemmers in NLTK

- Porter Stemmer
- Snowball Stemmer
- Lancaster Stemmer
- Regexp Stemmer

Porter Stemmer
It is a type of stemmer which is mainly known for Data Mining and Information Retrieval. As its applications are limited to the English language only. It is based on the idea that the suffixes in the English language are made up of a combination of smaller and simpler suffixes, it is also majorly known for its simplicity and speed. The advantage is, it produces the best output from other stemmers and has less error rate.

Implementation:
Use following libraries or functions to process accepted text.

- nltk
- nltk.tokenize
- WhitespaceTokenizer
- WordPunctTokenizer
- TreebankWordTokenizer
- TweetTokenizer
- MWETokenizer
- nltk.stem
- PorterStemmer
- SnowballStemmer
- LancasterStemmer
- RegexpStemmer
- WordNetLemmatizer

Conclusion : Executed tokenization using Use porter stemmer and snowball stemmer .

Experiment No: Group 1-2

Title: Perform bag-of-words approach, TF-IDF on data.

Objective: To Understand bag-of-words using Word2Vec .

Problem Statement: Perform bag-of-words approach (count occurrence, normalized count occurrence), TF-IDF on data. Create embeddings using Word2Vec. Dataset to be used: https://www.kaggle.com/datasets/CooperUnion/cardataset.

Theory:

What are Word Embeddings?
It is an approach for representing words and documents. Word Embedding or Word Vector is a numeric vector input that represents a word in a lower-dimensional space. It allows words with similar meaning to have a similar representation. They can also approximate meaning. A word vector with 50 values can represent 50 unique features.
Features: Anything that relates words to one another. Eg: Age, Sports, Fitness, Employed etc. Each word vector has values corresponding to these features.

- Implementations of Word Embeddings:

Word Embeddings are a method of extracting features out of text so that we can input those features into a machine learning model to work with text data. They try to preserve syntactical and semantic information. The methods such as Bag of Words(BOW), CountVectorizer and TFIDF rely on the word count in a sentence but do not save any syntactical or semantic information. In these algorithms, the size of the vector is the number of elements in the vocabulary. We can get a sparse matrix if most of the elements are zero. Large input vectors will mean a huge number of weights which will result in high computation required for training. Word Embeddings give a solution to these problems.

- Method I Word2vec:

Word2vec is one of the most popular technique to learn word embeddings using a two-layer neural network. Its input is a text corpus and its output is a set of vectors. Word embedding via word2vec can make natural language computer-readable, then further implementation of mathematical operations on words can be used to detect their similarities. A well-trained set of word vectors will place similar words close to each other in that space. For instance, the words women, men, and human might cluster in one corner, while yellow, red and blue cluster together in another.
There are two main training algorithms for word2vec, one is the continuous bag of words(CBOW), another is called skip-gram. The major difference between these two methods

is that CBOW is using context to predict a target word while skip-gram is using a word to predict a target context.

- Method II Bag of Words

Bag of words is a Natural Language Processing technique of text modelling. In technical terms, we can say that it is a method of feature extraction with text data. This approach is a simple and flexible way of extracting features from documents.

A bag of words is a representation of text that describes the occurrence of words within a document. We just keep track of word counts and disregard the grammatical details and the word order. It is called a "bag" of words because any information about the order or structure of words in the document is discarded. The model is only concerned with whether known words occur in the document, not where in the document.

One of the biggest problems with text is that it is messy and unstructured, and machine learning algorithms prefer structured, well defined fixed-length inputs and by using the Bag-of- Words technique we can convert variable-length texts into a fixed-length **vector.**

Implementation:

- Method I Word2vec:

Gensim Python Library Introduction

Gensim is an open source python library for natural language processing and it was developed and is maintained by the Czech natural language processing researcher Radim Řehůřek. Gensim library will enable us to develop word embeddings by training our own word2vec models on a custom corpus either with CBOW of skip-grams algorithms.

Steps in Implementation of Word Embedding with Gensim:

1. Install gensim
2. Download data (https://www.kaggle.com/datasets/CooperUnion/cardataset)
3. Data Preprocessing
4. Genism word2vec Model Training

- Method II Bag of Words: Techniques to build Bag of Words

1. Count Occurrence using CountVectorize
2. Normalized Count Occurrence using TfidfVectorizer
3. TF-IDF using TfidfVectorizer

Steps in Implementation of Bag of Words Techniques:

1. Download data (https://www.kaggle.com/datasets/CooperUnion/cardataset)
2. cleansing
3. Data Preprocessing
4. Model Building
5. pipeline

Conclusion : Performed bag-of-words approach, TF-IDF on data.

Experiment No: Group 1-3

Title: Perform text cleaning, perform lemmatization (any method), remove stop words (any method), label encoding.

Objective: To Understand Text Cleaning, lemmatization and label encoding using TF-IDF . .

Problem Statement: Perform text cleaning, perform lemmatization (any method), remove stop words (any method), label encoding. Create representations using TF-IDF. Save outputs. Dataset:https://github.com/PICT-NLP/BE-NLP-Elective/blob/main/3-Preprocessing/News_dataset.pickle

Theory:

In any machine learning task or data analysis task the first and foremost step is to clean and process the data. Cleaning is important for model building. Well, cleaning of data depends on the type of data and if the data is textual then it is more vital to clean the data.
Well, there are various types of text processing techniques that we can apply to the text data, but we need to be careful while applying and choosing the processing steps. Here, the steps of processing the textual data depend on the use cases.
For example, in sentiment analysis, we don't need to remove emojis or emoticons from the text as they convey the sentiment of the text. In this article, we will see some common methods and their code to clean the textual data.
Text Cleaning
Text cleaning is task-specific and one needs to have a strong idea about what they want their end result to be and even review the data to see what exactly they can achieve.
Take a couple of minutes and explore the data. What do you notice at a first glance? Here's what a trained I see:

- Having too many typos or spelling mistakes in the text
- Having too many numbers and punctuations (E.g. Love!!!!)
- Text is full of emojis and emoticons and username and links too. (If the text is from Twitter or Facebook)
- Some of the text parts are not in the English language. Data is having a mixture of more than one language
- Some of the words are combined with the hyphen or data having contractions words. (E.g. text-processing)
- Repetitions of words (E.g. Data)

Well, honestly there are many more things that a trained eye can see. But if we look in general and just want an overview then follow the article for it.
Most common methods for Cleaning the Data

$$w_{i,j} = tf_{i,j} \times idf_i$$

The TF-IDF score as the name suggests is just a multiplication of the term frequency matrix with its IDF, it can be calculated as follow:

Where w_i is TF-IDF score for term i in document j, tf_i is term frequency for term i in document j, and idf_i is IDF score for term i.

Implementation:

- Import required libraries
- Read dataset (Dataset: https://github.com/PICT-NLP/BE-NLP-Elective/blob/main/3-Preprocessing/News_dataset.pickle)
- Data Visualization using matplotlib to check the number of records as per categories (business, entertainment, political type, sport, tech)
- Identify stopwords from dataset
- Visualizing Stop Words in the dataset. Plot graph to show number of times occurrence of the stop words (the, an, a, in, to , of etc.)
- Text cleaning and basic preprocessing
  - Removing punctuation
  - Convert all letters in small case
  - Remove stop words
- Display clean dataset after performing cleaning operation and observe the change in the text
- Rearrange the columns after preprocessing to perform further operations.
- Extract data of “News”, “Type of news” and “Cleaned news”
- Perform lemaatization using WordNetLemmatizer
- Import “wordnet” to convert nltk tag to wordnet tag
- Perform label encoding on preprocessed dada using LabelEncoder
- Apply TFidf to perform vectorization.
- Save the output

Conclusion : Performed Text Cleaning, lemmatization and label encoding using TF-IDF . .

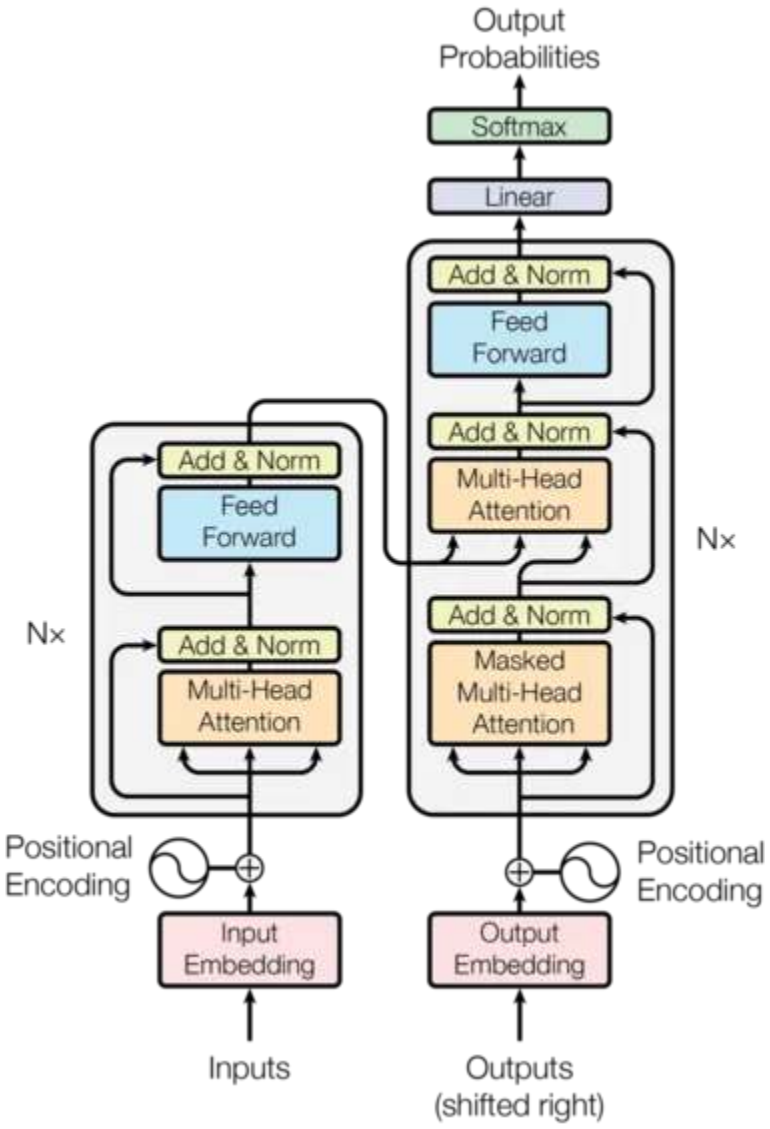
Experiment No: Group 1-4

Title: Transformer

Objective: To create a transformer. .

Problem Statement: Create a transformer from scratch using the Pytorch library

Theory:



In effect, there are five processes we need to understand to implement this model:

- Embedding the inputs
- The Positional Encodings
- Creating Masks
- The Multi-Head Attention layer
- The Feed-Forward layer

➢ Embedding

Embedding words has become standard practice in NMT, feeding the network with far more information about words than a one hot encoding would.

The embedding vector for each word will learn the meaning, so now we need to input something that tells the network about the word's position.

Vasmari et al answered this problem by using these functions to create a constant of position-specific values:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$

$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

This constant is a 2d matrix. *Pos* refers to the order in the sentence, and *i* refers to the position along the embedding vector dimension. Each value in the pos/i matrix is then worked out using the equations above.

➢ Creating Our Masks

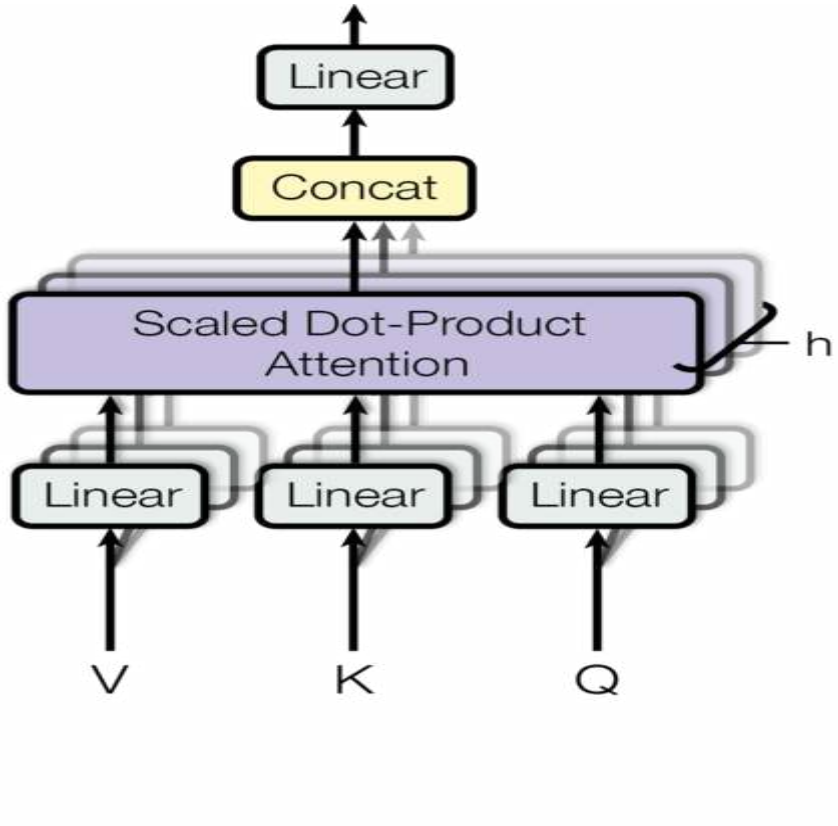
Masking plays an important role in the transformer. It serves two purposes:

- In the encoder and decoder: To zero attention outputs wherever there is just padding in the input sentences.
- In the decoder: To prevent the decoder 'peaking' ahead at the rest of the translated sentence when predicting the next word.

➢ Multi-Headed Attention

Once we have our embedded values (with positional encodings) and our masks, we can start building the layers of our model.

Here is an overview of the multi-headed attention layer:



$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Equation for calculating attention

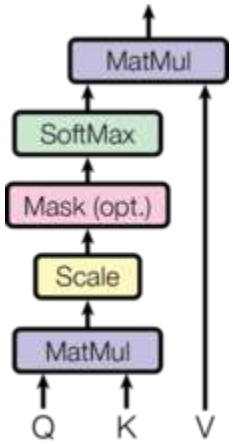


Diagram from paper illustrating equation steps

This is the only other equation we will be considering today, and this diagram from the paper does a god job at explaining each step.

Each arrow in the diagram reflects a part of the equation.

Initially we must multiply Q by the transpose of K. This is then ‘scaled’ by dividing the output by the square root of d_k.

Another step not shown is dropout, which we will apply after Softmax.

Finally, the last step is doing a dot product between the result so far and V.

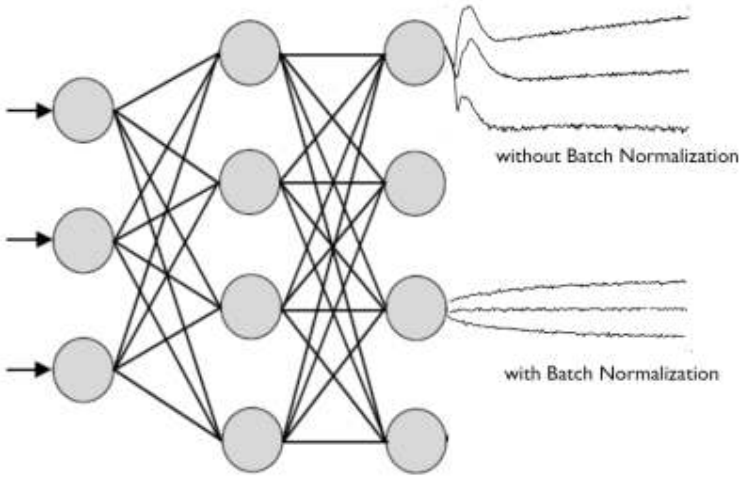
➢ The Feed-Forward Network:

This layer just consists of two linear operations, with a relu and dropout operation in between them.

The feed-forward layer simply deepens our network, employing linear layers to analyse patterns in the attention layers output.

One Last Thing : Normalisation

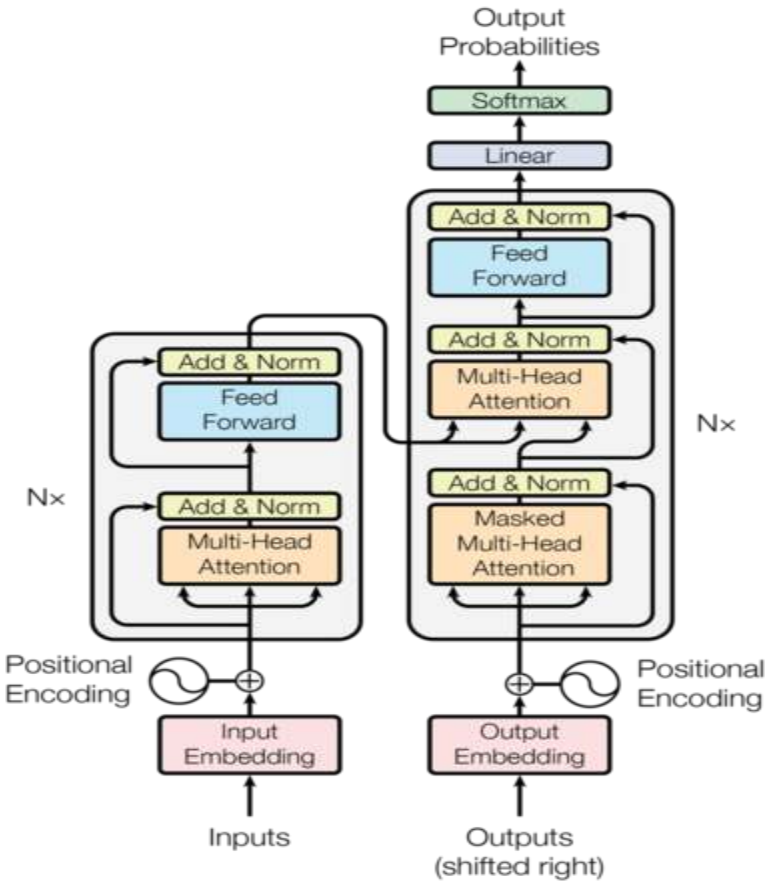
Normalisation is highly important in deep neural networks. It prevents the range of values in the layers changing too much, meaning the model trains faster and has better ability to generalise.



We will be normalising our results between each layer in the encoder/decoder.

If you understand the details above, you now understand the model. The rest is simply putting everything into place.

Let’s have another look at the over-all architecture and start building:



We will now build EncoderLayer and DecoderLayer modules with the architecture shown in the model above. Then when we build the encoder and decoder we can define how many of these layers to have.

➢ Training the model

With the transformer built, all that remains is to train that sucker on the EuroParl dataset. The coding part is pretty painless, but be prepared to wait for about 2 days for this model to start converging!

➢ Testing the model

We can use the below function to translate sentences. We can feed it sentences directly from our batches, or input custom strings.

The translator works by running a loop. We start off by encoding the English sentence. We then feed the decoder the <sos> token index and the encoder outputs. The decoder makes a prediction for the first word, and we add this to our decoder input with the sos token. We rerun the loop, getting the next prediction and adding this to the decoder input, until we reach the <eos> token letting us know it has finished translating.

Conclusion : Transformer created using Pytorch.

Experiment No: Group 1-5

Title: Morphology.

Objective: To understand the morphology of a word by the use of Add-Delete table..

Problem Statement: Morphology is the study of the way words are built up from smaller meaning bearing units. Study and understand the concepts of morphology by the use of add delete table.

Theory:

Morph Analyser
Definition
Morphemes are considered as smallest meaningful units of language. These morphemes can either be a root word(play) or affix(-ed). Combination of these morphemes is called morphological process. So, word "played" is made out of 2 morphemes "play" and "-ed". Thus finding all parts of a word(morphemes) and thus describing properties of a word is called "Morphological Analysis". For example, "played" has information verb "play" and "past tense", so given word is past tense form of verb "play".
Analysis of a word :
बच्चों (bachchoM) = बच्चा(bachchaa)(root) + ओं(oM)(suffix) (ओं=3 plural oblique) A linguistic paradigm is the complete set of variants of a given lexeme. These variants can be classified according to shared inflectional categories (eg: number, case etc) and arranged into tables.
Paradigm for बच्चा

Case/num	Singular	Plural
Direct	बच्चा(bachchaa)	बच्चे(bachche)
oblique	बच्चे(bachche)	बच्चों (bachchoM)

Algorithm to get बच्चच(bachchoM) from बच्चा(bachchaa)

1. Take Root बच्च(bachch)आ(aa)
2. Delete आ(aa)
3. output बच्च(bachch)
4. Add ओं(oM) to output

5. Return बच्चों (bachchoM)
Therefore आ is deleted and ओं is added to get बच्चों
Add-Delete table for बच्चा

Delete	Add	Number	Case	Variants
आ(aa)	आ(aa)	sing	dr	बच्चा(bachchaa)
आ(aa)	ए(e)	Plu	dr	बच्चे(bachche)
आ(aa)	ए(e)	Sing	ob	बच्चे(bachche)
आ(aa)	ओं(oM)	Plu	ob	बच्चों(bachchoM)

Paradigm Class
Words in the same paradigm class behave similarly, for Example लड़क is in the same paradigm class as बच्च, so लड़का would behave similarly as बच्चा as they share the same paradigm class.
STEP 1: Select a word root.
STEP 2: Fill the add-delete table and submit.
STEP 3: If wrong, see the correct answer or repeat STEP1.

Let's consider the word "unhappily". We can break this word down into smaller units or morphemes:

Morpheme	**Meaning**	**Example**
un-	not	unhappy

-happi-	happy	happy
-ly	adverb	happily

Using an add-delete table, we can see how these morphemes combine to form the word "unhappily":

Add	**Delete**	**Intermediate Forms**
	unhappily	
un-	happily	unhappily
	happi-ly	unhappy-ly

Morphology is important in understanding how words are formed and how their meanings can change through the addition or deletion of morphemes. By breaking down words into their morphemes, we can also understand the grammatical structure of the language and how different parts of speech are formed.

Conclusion : We have Studied and understood the concepts of morphology by the use of add delete table.

Experiment No: Group 2-7

Title: POS (Part-of-Speech) tagging.

Objective: To assign the correct grammatical tags to each word in a sentence, based on their role and context in the sentence.

Problem Statement: POS Taggers For Indian Languages.

Theory:

POS (Part-of-Speech) tagging is the process of marking each word in a text with its corresponding part of speech, such as noun, verb, adjective, adverb, etc. POS tagging is an important task in natural language processing and is used in many applications such as machine translation, speech recognition, and text-to-speech systems.

In Indian languages, POS tagging is a challenging task due to the complex morphology and rich inflectional and derivational systems. Moreover, the lack of standardization and uniformity in the use of scripts, orthography, and grammar across different regions and dialects makes it even more challenging.

Several approaches have been proposed for POS tagging in Indian languages. These include rule-based, dictionary-based, and machine learning-based methods. Rule-based methods rely on a set of handcrafted rules that define the patterns of each part of speech. Dictionary-based methods use a pre-defined dictionary of words and their corresponding parts of speech. Machine learning-based methods use annotated training data to learn the statistical patterns and relationships between words and their parts of speech.

One popular machine learning-based approach for POS tagging in Indian languages is the Hidden Markov Model (HMM). In this approach, each word is modeled as a sequence of hidden states, where each state corresponds to a part of speech. The model is trained on annotated data using the Baum-Welch algorithm to estimate the transition probabilities between states and the emission probabilities of each word given its state. The Viterbi algorithm is then used to find the most likely sequence of states (i.e., the most likely part-of-speech tags) for a given sentence.

Another popular machine learning-based approach for POS tagging in Indian languages is the Conditional Random Field (CRF) model. In this approach, the sequence of words and their corresponding part-of-speech tags is modeled as a graph, where the nodes represent the words and the edges represent the transition probabilities between adjacent tags. The model is trained on annotated data using the maximum likelihood estimation algorithm to learn the weights of the features that capture the statistical patterns and dependencies between the words and their

tags. The Viterbi algorithm is then used to find the most likely sequence of tags for a given sentence.

Overall, POS tagging in Indian languages is a challenging task due to the complexity and diversity of the languages. However, with the availability of large annotated datasets and advanced machine learning techniques, accurate and efficient POS taggers can be developed for Indian languages, which can help in improving the performance of various NLP applications.

Another approach for POS tagging in Indian languages is to use statistical taggers. These taggers use machine learning algorithms to learn the patterns and structures of the language from a large corpus of annotated data. Once trained, the tagger can then tag new sentences with high accuracy. However, the availability of large annotated corpora is a major challenge in developing statistical taggers for Indian languages.

Some examples of POS taggers for Indian languages are:

1. NLTK - The Natural Language Toolkit (NLTK) provides POS taggers for several Indian languages including Hindi, Bengali, Marathi, Telugu, and Urdu. These taggers are based on a combination of rule-based and statistical techniques.
2. Stanford NLP - The Stanford NLP toolkit provides POS taggers for several Indian languages including Hindi, Bengali, and Telugu. These taggers are based on statistical techniques and have achieved high accuracy on benchmark datasets.
3. ILMT - The Indian Language Machine Translation (ILMT) project provides POS taggers for several Indian languages including Hindi, Bengali, and Tamil. These taggers are based on rule-based techniques and have been used in several machine translation applications.

In conclusion, POS tagging is an important task in NLP and there are several POS taggers available for Indian languages that can accurately tag words in a sentence. These taggers use a combination of rule-based and statistical techniques and have been developed for several Indian languages. However, the availability of annotated corpora is a major challenge in developing accurate POS taggers for Indian languages.

Conclusion : Performed POS Taggers For Indian Languages like Hindi, Bengali etc.