Dr. D. Y. Patil Pratishthan's

DR. D. Y. PATIL INSTITUTE OF ENGINEERING, MANAGEMENT & RESEARCH

Approved by A.I.C.T.E, New Delhi , Maharashtra State Government, Affiliated to Savitribai Phule Pune University
Sector No. 29, PCNTDA , Nigidi Pradhikaran, Akurdi, Pune 411044. Phone: 020–27654470, Fax: 020-27656566
Website : www.dypiemr.ac.in Email : principal.dypiemr@gmail.com

---

DEPARTMENT
OF
COMPUTER ENGINEERING

LAB MANUAL
Lab Practices VI
(Final Year Engineering)
Semester – II

Prepared by : Mrs. P P Shevatekar
Dr Nalini Jagtap

Institute Vision

To strive for excellence by providing quality technical education and facilitate research for the welfare of society

Institute Mission

1. To educate students with strong fundamentals by providing conducive environment
2. To inculcate research with creativity & innovation
3. To strengthen leadership, team work, professional & communication skills and ethical standards
4. To promote Industry Institute collaboration & prepare students for life long learning in context of technological change.

Department Vision

To produce quality computer professionals and fostering research aptitude for dispensing service to society

Department Mission

1. To promote growth of an individual by imparting comprehensive knowledge of tools and technologies.
2. To facilitate research and innovation by engaging faculty and students in research activities.
3. To enrich industry-institute interaction in order to provide a platform to know industry demands and motivation for self-employment.
4. To bring forth a conducive environment to enhance soft skills and professional skills to cater needs of society

Program Specific Outcomes

1. Professional Skills: The ability to comprehend, analyze and develop software and hardware systems and amp; applications through research, in varying domains.

2. Problem-Solving Skills: The ability to apply standard paradigms and strategies in software project development using open-ended programming environments to deliver a quality product.

3. Successful Career and Entrepreneurship: Adaptation of modern practical and systematic approaches in creating innovative solutions for a successful career, entrepreneurship, and a zest for higher studies.

Course Objective

1. To understand the fundamental concepts and techniques of natural language processing (NLP) .
2. To understand Digital Image Processing Concepts .
3. To learn the fundamentals of software defined networks .
4. Explore the knowledge of adaptive filtering and Multi-rate DSP To be familiar with the various application areas of soft computing.
5. To introduce the concepts and components of Business Intelligence (BI) .
6. To study Quantum Algorithms and apply these to develop hybrid solutions.

Course Outcome

On completion of this course, the students will be able to

CO1: Apply basic principles of elective subjects to problem solving and modeling.

CO2: Use tools and techniques in the area of software development to build mini projects.

CO3: Design and develop applications on subjects of their choice.

CO4: Generate and manage deployment, administration & security.

Guidelines for Students

The laboratory assignments are to be submitted by students in the form of a journal. Journal may consists of prologue, Certificate, table of contents, and handwritten write-up of each assignment (Title, Objectives, Problem Statement, Outcomes, software and Hardware requirements, Date of Completion, Assessment grade/marks and assessor's sign, Theory- Concept in brief, Algorithm/Database design, test cases, conclusion/analysis). Program codes with sample output of all performed assignments are to be submitted as softcopy. As a conscious effort and little contribution towards Green IT and environment awareness, attaching printed papers as part of write-ups and program listing to journal may be avoided. Use of digital storage media/DVD containing students programs maintained by lab In-charge is highly encouraged. For reference one or two journals may be maintained with program prints at Laboratory.

Guidelines for Laboratory /Term Work Assessment

Continuous assessment of laboratory work is to be done based on overall performance and lab Home Faculty of Engineering Savitribai Phule Pune University Syllabus for Fourth Year of Computer Engineering assignments performance of student. Each lab assignment assessment will assign grade/marks based on parameters with appropriate weightage. Suggested parameters for overall assessment as well as each lab assignment assessment include- timely completion, performance, innovation, efficient codes, punctuality and neatness reserving weightage for successful mini-project completion and related documentation.

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.
Tokenization converts an unstructured string (text document) into a numerical data structure suitable for machine learning. This allows the machines to understand each of the words by themselves, as well as how they function in the larger text. This is especially important for larger amounts of text as it allows the machine to count the frequencies of certain words as well as where they frequently appear.
Tokenization is the first crucial step of the NLP process as it converts sentences into understandable bits of data for the program to work with. Without a proper / correct tokenization, the NLP process can quickly devolve into a chaotic task.

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.

- In languages like English or French we can separate words by using white spaces, or punctuation marks to define the boundary of the sentences. But this method is not applicable for symbol based languages like Chinese, Japanese, Korean Thai, Hindi, Urdu, Tamil, and others. Hence a common tokenization tool that combines all languages is needed.

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.

Why is stemming required?

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.
To build a robust model, it is essential to normalize text by removing repetition and transforming words to their base form through stemming.

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.

Lemmatization
Use any technique for lemmatization.
Lemmatization, unlike stemming reduces the inflected words properly ensuring that the root word belongs to the language.
Lemmatization is the grouping together of different forms of the same word. In search queries, lemmatization allows end users to query any version of a base word and get relevant results. Because search engine algorithms use lemmatization, the user is free to query any inflectional form of a word and get relevant results. For example, if the user queries the plural form of a word (e.g., routers), the search engine knows to also return relevant content that uses the singular form of the same word (router).
Lemmatization is extremely important because it is far more accurate than stemming. This brings great value when working with a chatbot where it is crucial to understand the meaning of a user's messages.
The major disadvantage to lemmatization algorithms, however, is that they are much slower than stemming algorithms.

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.
Generally, the skip-gram method can have a better performance compared with CBOW method, for it can capture two semantics for a single word. For instance, it will have two vector representations for Apple, one for the company and another for the fruit.

- 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.
Why is the Bag-of-Words algorithm used?
So, why bag-of-words, what is wrong with the simple and easy text?
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.**
Also, at a much granular level, the machine learning models work with numerical data rather than textual data. So to be more specific, by using the bag-of-words (BoW) technique, we convert a text into its equivalent vector of numbers.

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
We will see how to code and clean the textual data for the following methods.

- Lowecasing the data
- Removing Puncuatations
- Removing Numbers
- Removing extra space
- Replacing the repetitions of punctations
- Removing Emojis
- Removing emoticons
- Removing Contractions Importing the library

```
import pandas as pd
##Let's read the sample data
text = "I had such high hopes for this dress 15 size or (my usual size) to work for me."
print(text)
##Output
I had such high hopes for this dress 15 size or (my usual size) to work for me.
```

- Lower Casing the Data

From the first glance we just lower case the data. The idea is to convert the input text into the same casing format so that it converts 'DATA', 'Data', 'DaTa', 'DATa' into 'data'.

In some use cases, like the tokenizer and vectorization processes, the lower casing is done beforehand. But choose the lower casing precisely because if we are doing sentiment analysis on the text then if we make the text in lower case then sometimes we might miss what the word is actually stating. For example, if the word is in the upper case then it refers to anger and so on.

Here, for lower casing the data we will use the lower() method to convert all the text into one common lower format.

```
ans = text.lower() ans
#Output
'i had such high hopes for this dress 15 size or (my usual size) to work for me.'
```

- Removing Punctuations

The second most common text processing technique is removing punctuations from the textual data. The punctuation removal process will help to treat each text equally. For example, the word data and data! are treated equally after the process of removal of punctuations.

We need to take care of the text while removing the punctuation because the contraction words will not have any meaning after the punctuation removal process. Such as 'don't' will convert to 'dont' or 'don t' depending upon what you set in the parameter.

We also need to be extra careful while choosing the list of punctuations that we want to exclude from the data depending upon the use cases. As string.punctuation in python contains these symbols !"#$%&\'()*+,-./:;?@[\\]^_{|}~`

```
import string
text = "I had such high hopes! for this dress size or (my usual size) to work for me."
PUNCT_TO_REMOVE = string.punctuation
ans = text.translate(str.maketrans(", ",
PUNCT_TO_REMOVE)) ans
```

#Output
#'I had such high hopes for this dress 15 size or my usual size to work for me'

- Removing Numbers

Sometimes number doesn't hold any vital information in the text depending upon the use cases. So it is better to remove them than to keep them.
For example, when we are doing sentiment analysis then the number doesn't hold any specific meaning to the data but if the task is to perform NER (Name Entity Recognition) or POS (Part of Speech tagging) then use the removing of number technique carefully.
Here, we are using the isdigit() function to see if the data has a number in it or not, and if we encountered the number then we are replacing the number with the blank.

```
ans = ''.join([i for i in text if not
i.isdigit()]) ans
```

#Output
'I had such high hopes for this dress size or (my usual size) to work for me."

- Removing Extra Space

Well, removing the extra space is good as it doesn't store extra memory and even we can see the data clearly.

```
ans = ''
''.join(text.split()) ans
```

#Output
'I had such high hopes for this dress 15 size or (my usual size) to work for me.'

- Replacing the Repetitions of Punctuations

Having knowledge of regular expression will help to code faster and easier. To remove the repetition of punctuations is very helpful because it doesn't hold any vital information if we keep more than one punctuation in the word, for example, data!!! need to convert to data.
Let's first see how to replace the repetitions of punctuations. Here, we are replacing the word dress!!!! to dress and just replacing one punctuation only.

```
text1 = "I had such... high hopes for this dress!!!!"
ans = re.sub(r'(!)1+', '', text1)
```

- ans = re.sub(r'(!)1+', '', text1)
- ans #Output
- 'I had such... high hopes for this dress'
- What if the text has more than just one punctuation in them let's look at the below example to understand it.
- import re
- text1 = "I had such... high hopes for this dress!!!!" ans = re.sub(r'(!|.)1+', '', text1)
- ans #Output
- 'I had such high hopes for this dress'

'Removing Emojis
Growing users of the audience on the social media platforms, well there is a significant explosion of usage of emojis in day-to-day life. Well, when we are performing text analysis in

some cases removal of emojis is the correct way as sometimes they don't hold any information.

Below is the helper function from which the emojis will be replaced with the blank.

```
def remove_emoji(string):
emoji_pattern = re.compile("["u"U0001F600-U0001F64F" # emoticons
u"U0001F300-U0001F5FF" # symbols & pictographs
u"U0001F680-U0001F6FF" # transport & map symbols
u"U0001F1E0-U0001F1FF" # flags (iOS) u"U00002702-U000027B0"
u"U000024C2-U0001F251"
"]+", flags=re.UNICODE) return
emoji_pattern.sub(r'', string)

remove_emoji("game is on
") #Output
'game is on '
```

- Removing Emoticons

While doing the text analysis of Twitter and Instagram data we often find this emoticon and nowadays, there is hardly any text which doesn't contain any emoticons in them.

The below helper function help to remove the emoticons from the text. The EMOTICIONS dictionary consists of the symbols and names of the emoticons you can customize the EMOTICONS as per your need.

```
# Removing of Emoticons
EMOTICONS = {
u":-)":"Happy face or smiley",
u":)":"Happy face or smiley",
u":-]":"Happy face or smiley",
u":]":"Happy face or smiley",
u":-3″:"Happy face smiley",
u":3″:"Happy face smiley",
u":->":"Happy face smiley",
u":>":"Happy face smiley",
u"8-)":"Happy face smiley",
u":o)":"Happy face smiley",
u":-}":"Happy face smiley",
u":}":"Happy face smiley",
u":-)":"Happy face smiley",
u":c)":"Happy face smiley",

u":^)":"Happy face smiley",
u"=]":"Happy face smiley"
}
text = 'I had such high hopes for this dress 15 size really wanted it to work for me :-)' ans
= re.compile(u'(' + u'|'.join(k for k in EMOTICONS) + u')')
```

```
ans = ans.sub(r",text)
ans


#Output
#I had such high hopes for this dress 15 size really wanted it to work for me
```

The code of Removal of emoticons is taken from: here

While sometimes we don't want the emoticons so, we remove them but what if I say there is a way around it. Let's see if we remove the emotions and put alternative words, for example, removing this ":-)" emoticon and replacing it with text such as Happy face smiley or any custom name you like.

```
def convert_emoticons(text):
for emot in EMOTICONS:
text = re.sub(u'('+emot+')', "_".join(EMOTICONS[emot].replace(",","").split()), text)
return text
text = "Hello :-)"
convert_emoticons(text)
#Output
'Hello Happy_face_smiley'
```

- Removing Contractions

There are so many contractions in the text we type so to expand them we will use the contractions library.

The Twitter and Instagram data has so many contractions in them and if we remove the punctuations from that text then it would look like this.

For example, the text "I'll eat pizza" and if we remove the punctuations them the text will look like this "I ll will eat pizza". Here, "I ll" doesn't hold any information to the text that's why we use the contraction.

Importing the library

```
!pip install contractions
```

Let's see how it's done.

```
import contractions
text = "She'd like to know how I'd do that!"
contractions.fix(text)
#Output
she would like to know how I would do that!
```

- Wordnet:

WordNET is a lexical database of words in more than 200 languages in which we have adjectives, adverbs, nouns, and verbs grouped differently into a set of cognitive synonyms, where each word in the database is expressing its distinct concept. The cognitive synonyms which are called synsets are presented in the database with lexical and semantic relations. WordNET is publicly available for download and also we can test its network of related words and concepts using this.

- Label encoder

Label Encoding cites the transmogrification of the labels into the numeric form to change it into a form that can be read by the machine. Machine learning algorithms can thereafter determine in a correct way as to how these labels must be managed. It is a crucial pre-processing measure during the integrated dataset in supervised learning.
For example, we have a dataset that has a comparison of a certain quality in a certain skill in the form of a superlative comparison between siblings. The dataset is good, better, best. After applying a label encoder each quality will be given a label 0,1,2 respectively. The label for good quality is 0, for better the label is 1, and for best quality, the label is 2.
The above-mentioned example was basic in terms of the dataset. The conversion can be of any dataset be it of height, age, eye colour, iris type, symptoms, etc.
Label Encoding in Python can be implemented using the Sklearn Library. Sklearn furnishes a very effective method for encoding the categories of categorical features into numeric values. Label encoder encodes labels with credit between 0 and n-1 classes where n is the
number of diverse labels. If a label reiterates it appoints the exact merit to as appointed before.

And to renovate this type of categorical text data into data that can be understood by model numerical data, we use the Label Encoder class. We need to label encode the initial column, import the LabelEncoder class from the sklearn library, equip and revamp the initial section of the data, and then rehabilitate the occurring text data with the fresh encoded data.

- TF-IDF

Term frequency-inverse document frequency is a text vectorizer that transforms the text into a usable vector. It combines 2 concepts, Term Frequency (TF) and Document Frequency (DF). The term frequency is the number of occurrences of a specific term in a document. Term frequency indicates how important a specific term in a document. Term frequency represents every text from the data as a matrix whose rows are the number of documents and columns are the number of distinct terms throughout all documents.
Document frequency is the number of documents containing a specific term. Document frequency indicates how common the term is.
Inverse document frequency (IDF) is the weight of a term, it aims to reduce the weight of a term if the term's occurrences are scattered throughout all the documents. IDF can be calculated as follow:

$$idf_i = \log(\frac{n}{df_i})$$

Where idf_i is the IDF score for term i, df_i is the number of documents containing term i, and n is the total number of documents. The higher the DF of a term, the lower the IDF for the term.
When the number of DF is equal to n which means that the term appears in all documents, the IDF will be zero, since log(1) is zero, when in doubt just put this term in the stopword list because it doesn't provide much information.

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

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 . .

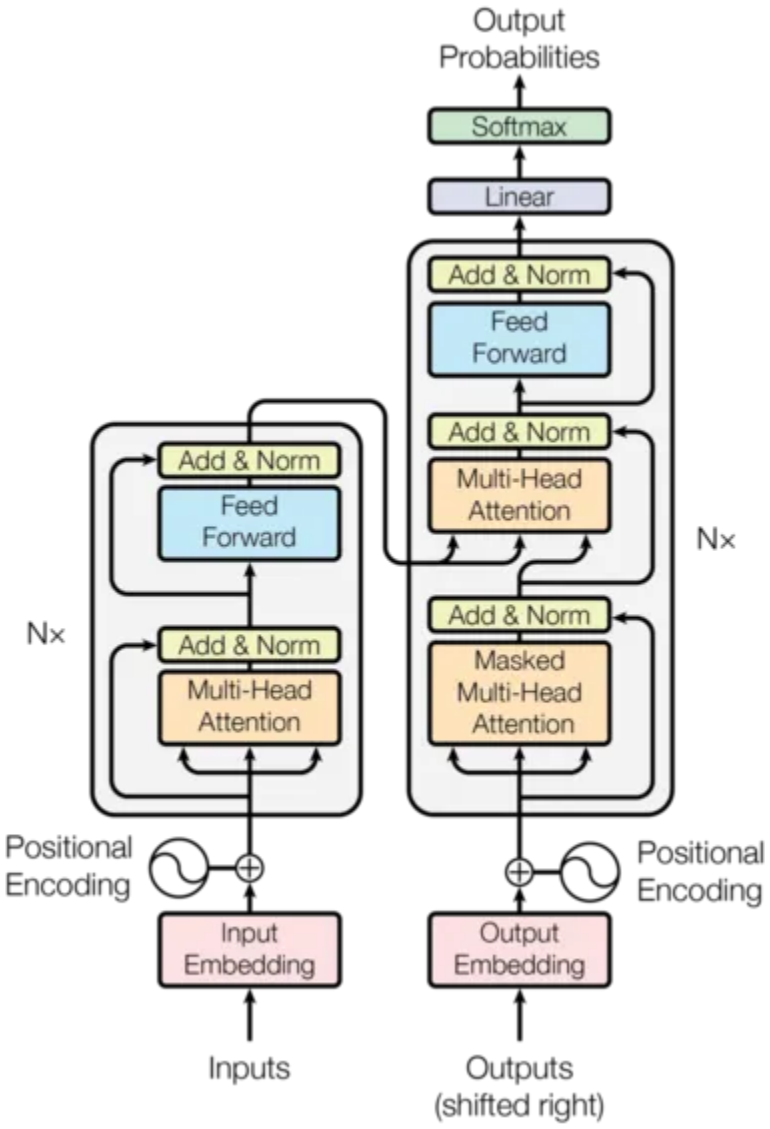
Experiment No: Group 1-4

Title: Transformer

Objective: To create a transformer. .

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

Theory:



The diagram above shows the overview of the Transformer model. The inputs to the encoder will be the English sentence, and the 'Outputs' entering the decoder will be the French sentence.

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.

When each word is fed into the network, this code will perform a look-up and retrieve its embedding vector. These vectors will then be learnt as a parameters by the model, adjusted with each iteration of gradient descent.Giving our words context: The positional encoding

In order for the model to make sense of a sentence, it needs to know two things about each word: what does the word mean? And what is its position in the sentence?

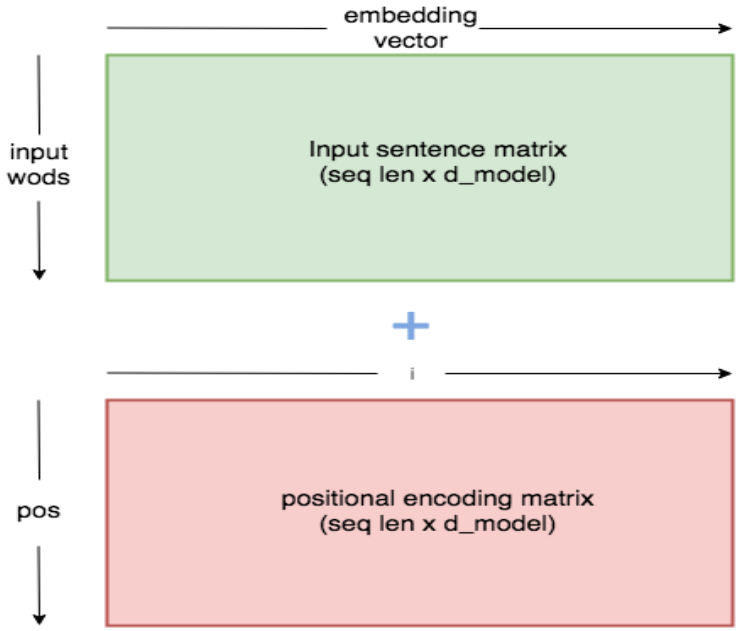
The embedding vector for each word will learn the meaning, so now we need to input something that tells the network about the word's position.

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

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

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

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.



The positional encoding matrix is a constant whose values are defined by the above equations. When added to the embedding matrix, each word embedding is altered in a way specific to its position.

The above module lets us add the positional encoding to the embedding vector, providing information about structure to the model.

The reason we increase the embedding values before addition is to make the positional encoding relatively smaller. This means the original meaning in the embedding vector won't be lost when we add them together.

➢ 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.

The initial input into the decoder will be the target sequence (the French translation). The way the decoder predicts each output word is by making use of all the encoder outputs and the French sentence only up until the point of each word its predicting.

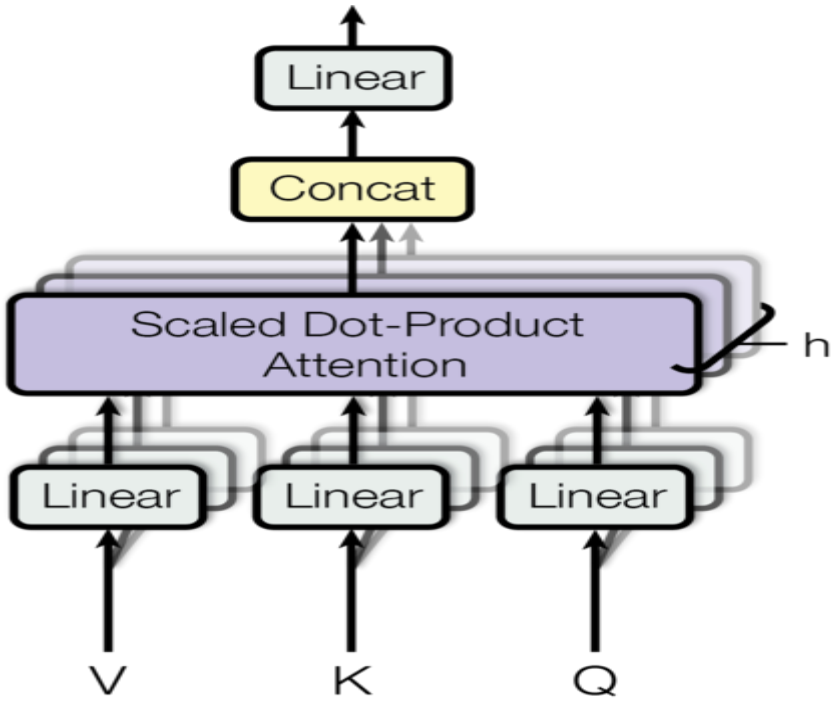
Therefore we need to prevent the first output predictions from being able to see later into the sentence. For this we use the nopeak_mask

If we later apply this mask to the attention scores, the values wherever the input is ahead will not be able to contribute when calculating the outputs.

➢ 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:



Multi-headed attention layer, each input is split into multiple heads which allows the network to simultaneously attend to different subsections of each embedding.

V, K and Q stand for 'key', 'value' and 'query'. These are terms used in attention functions, but honestly, I don't think explaining this terminology is particularly important for understanding the model.

In the case of the Encoder, *V, K* and *G* will simply be identical copies of the embedding vector (plus positional encoding). They will have the dimensions Batch_size * seq_len * d_model.

In multi-head attention we split the embedding vector into *N* heads, so they will then have the dimensions batch_size * N * seq_len * (d_model / N).

This final dimension (d_model / N) we will refer to as d_k.Calculating Attention

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

Equation for calculating attention

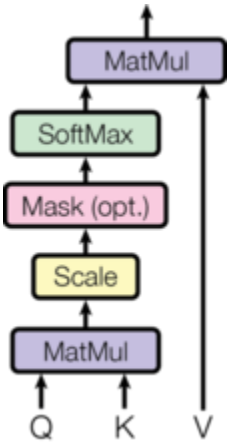


Diagram from paper illustrating equation steps

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.

A step that's not shown in the equation is the masking operation. Before we perform Softmax, we apply our mask and hence reduce values where the input is padding (or in the decoder, also where the input is ahead of the current word).

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:

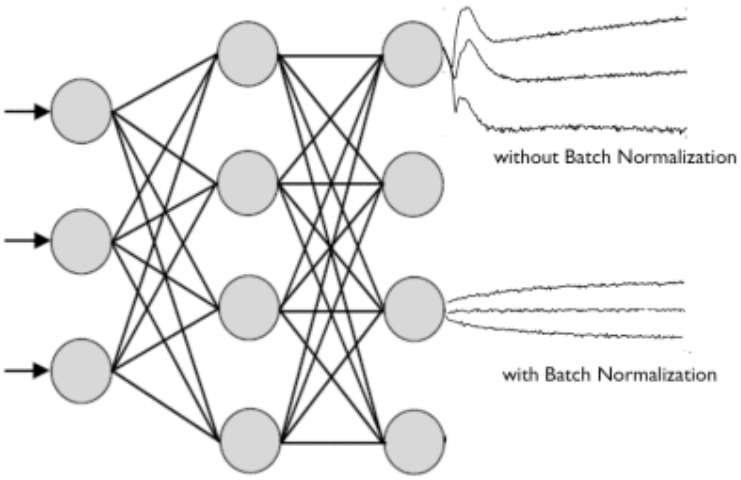
Ok if you've understood so far, give yourself a big pat on the back as we've made it to the final layer and it's all pretty simple from here!

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

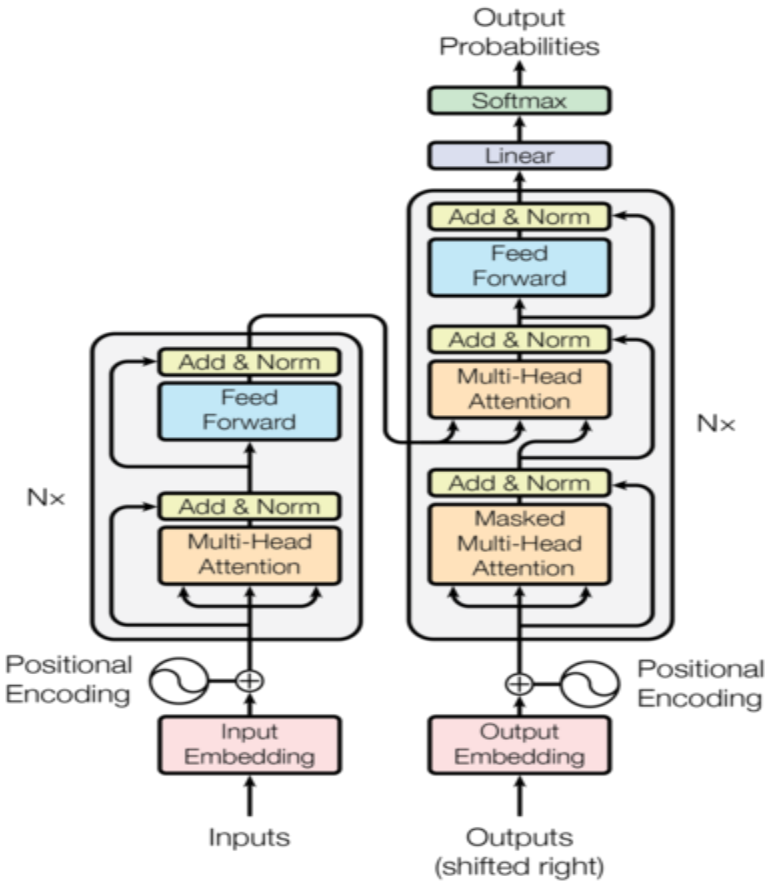
Normalisation is highly important in deep neural networks. It prevents the range of values in the layers changing too much, meaning the model trains faster and has better ability to generalise.



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:



One last Variable: If you look at the diagram closely you can see a 'Nx' next to the encoder and decoder architectures. In reality, the encoder and decoder in the diagram above represent one layer of an encoder and one of the decoder. N is the variable for the number of layers there will be. Eg. if N=6, the data goes through six encoder layers (with the architecture seen above), then these outputs are passed to the decoder which also consists of six repeating decoder layers.

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 बच्चों(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
अ (aa)	आ (aa)	sing	dir	बच्चा(bachchaa)
अ (aa)	ए(e)	Plu	dir	बच्चे(bachche)
अ (aa)	ए(e)	Sing	obl	बच्चे(bachche)
अ (aa)	ओं (oM)	Plu	obl	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

In this example, the morpheme "un-" is added to the word "happy" to form "unhappy". Then, the morpheme "-ly" is added to "unhappy" to form "unhappily".

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.

- **Oral Questions**:

1.Select words from this which belong to the same paradigm: मनुष्य(manuShya), पक्षी(pakshii), शिशु(shishu), गुरु(guruu), नर(nar)

2. Construct the paradigm table for the above words.

3. Observe the following words from Bengali. Identify all the morphemes and their corresponding meanings.

- kori '(I)do'
- maari '(I) hit'
- korchille '(You) were doing'
- maar '(You) hit'

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.