Summer Of Science

Natural Language Proccessing

MENTOR – OM KULPE

MENTEE – HARESH GUPTA

Roll NO. – **23b0931**

Introduction to Python Basic :

Python a interpreted language is a very friendly programming language and very easy to learn .

Basic of python which should be learned are :-

- Getting familiar with basic syntax of python
- Getting used to the syntax of logical part (if-else , switch) part of python
- Other than that getting used to python modules like numpy , matplotlib and pandas .

What is Natural Language Proccessing ?

By "natural language" we mean a language that is used for everyday communication by humans; languages such as English, Hindi, or Portuguese. In contrast to artificial languages such as programming languages and mathematical notations, natural languages have evolved as they pass from generation to generation, and are hard to pin down with explicit rules. We will take Natural Language Processing in a wide sense to cover any kind of computer manipulation of natural language. At one extreme, it could be as simple as counting word frequencies to compare different writing styles. At the other extreme, NLP involves "understanding" complete human utterances, at least to the extent of being able to give useful responses to them. Technologies based on NLP are becoming increasingly widespread. For example, phones and handheld computers support predictive text and handwriting recognition; web search engines give access to information locked up in unstructured text; machine translation allows us to retrieve texts written in Chinese and

read them in Spanish. By providing more natural human-machine interfaces, and more sophisticated access to stored information, language processing has come to play a central role in the multilingual information society

Application of Natural Language Proccessing :

1. Chatbots : Chatbots are form of AI that are programmed to interact with humans in such a way that they sound like humans themselves . Depending on complexities , they can either just respind to some specific keywords or they can understand the whole sentence just like humans , such bots improve themselves with time .
   Ex : It can be found in almost all of the app now-a-days like amazon , jiomart etc .

2. Autocomplete in search Engines : Search Engines tend to guess what are you typing and automatically complete your sentences . On typing "3blue" it will complete it as "3blue1Brown" , All these suggestions are provided using autocomplete that uses NLP to guess what you want to ask . They use NLP to make sense of these words and how they are interconnected to form different sentences.

    a. Voice Assistants : These days voice assistants are very popular and their capability to interpret natural languages are increasing exponentially . They use a complex combination of speech recognition , natural language understanding , and NLP to understand what humans are trying to say and then act on it . The long term goal of voice assistants is to become

a bridge between humans and internet and provide all manner of services based on just voice interaction .

Text Preprocessing :

Why is text preprocessing is important ?

The raw Text Data which we use as input is often noisy and unstructured , consisting various inconsistencies such as abbreviations , slang and irrelevant information . Preprocessing helps in :

1. Improving data quality : Removing Noise and irrelevant information ensures that data fed into the model is clean and consistent
2. Reducing Complexity : Simplifying the text data can reduce the computational complexity and make model more efficients.
3. Enhancing Model Performance : Training on good quality of data without noise would improve the performance of NLP models .

1.Following python codes demonstrates how to clean the data :

```python

import re
import string

def PreProccesText(text) :
    text = text.lower() # Conerting in to lowercase
    text = re.sub(r'\d+','',text) # Remove numbers
    text = text.translate(str.maketrans('','',string.punctuation)) # Removing Punctuations from data
    text = re.sub(r'\W','',text) # Remove special characters
    return text

text = ["Hello haresh here ", "what's up all good" , " btw you were asking about email it is 'haresh2op@gmail.com'"]

preProcces_text = [PreProccesText(sentences) for sentences in text]

print(preProcces_text)
```

Output of the above code :

```
                        > python -u "c:\Users\hp\Desktop\C++\Dynamic Programming\tpp.p
y"
['hello haresh here ', 'whats up all good', ' btw you were asking about email it is hare
shopgmailcom']
```

2.Following code demonstrates how to perform tokenization of the data and how to remove common stop words from the tokens :

```
18  import nltk
19  from nltk.tokenize import word_tokenize
20  from nltk.corpus import stopwords
21
22  nltk.download('punkt')
23  nltk.download('stopwords')
24
25  tokenized_text = [word_tokenize(sentence) for sentence in
    preProcces_text]
26
27  print(tokenized_text)
28
29  stop_words = set(stopwords.words('english'))
30
31  filtered_text = [[word for word in sentence if word not in stop_words]
    for sentence in tokenized_text]
32
33  print(filtered_text)
34
```

Output :

```
                        > python -u "c:\Users\hp\Desktop\C++\Dynamic Programming\tpp.p
y"
['hello haresh here ', 'whats up all good', ' btw you were asking about email it is hare
shopgmailcom']
[nltk_data] Downloading package punkt to
[nltk_data]     C:\Users\hp\AppData\Roaming\nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     C:\Users\hp\AppData\Roaming\nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[['hello', 'haresh', 'here'], ['whats', 'up', 'all', 'good'], ['btw', 'you', 'were', 'as
king', 'about', 'email', 'it', 'is', 'hareshopgmailcom']]
[['hello', 'haresh'], ['whats', 'good'], ['btw', 'asking', 'email', 'hareshopgmailcom']]
PS C:\Users\hp\Desktop\C++>
```

4.Handling Contraction : Following code expands the contraction in the text

```
36
37    import contractions
38
39    expanded_text = [contractions.fix(sentence) for sentence in
      preProcces_text]
40
41    print(expanded_text)
42
43
```

Output :

```
['hello haresh here ', 'what is up all good', ' by the way you were asking about email i
t is hareshopgmailcom']
```

5.Handling Emojis and Emoticons : Converting emojis to their textual representation (The output wouldn't be changed as my text didn't have any emojis)

```
43
44    import emoji
45
46    emoji_text = [emoji.demojize(sentence) for sentence in preProcces_text]
47
48    print(emoji_text)
49
```

Output :

```
['hello haresh here ', 'what is up all good', ' by the way you were asking about email i
t is hareshopgmailcom']
```

The above pieces of code gives an idea of how text Preproccessing is done. Also other than these there are more level of abstraction to improve the quality of raw data which is not covered in my code . For example spell check . This was all about text preprocessing

Part-of-speech tagging :

POS tagging is the process of assigning parts of speech to words in a text, such as nouns, verbs, adjectives, adverbs, etc.

It entails the assignment of a label to every word in a sentence indicating its grammatical category and function in that sentence.

POS tagging is crucial for natural language processing to analyze and comprehend textual data structure and meaning.

Role of Python in POS tagging :

Python libraries as NLTK offer tools and functionalities for conducting part of speech (POS) tagging on data.

In Python it is possible to create POS taggers that assign tags to words automatically by considering their context within a sentence.

Using Python programmers can tokenize text effortlessly handle word sequences and utilize dictionaries to associate words with their corresponding parts of speech tags. This simplifies the process of performing POS tagging tasks in natural language processing.

Using a Tagger :

From the NLTK library we can use the pos_tag method to tag a string

Following code shows how we can tag a string :

```
64  import nltk
65  nltk.download('averaged_perceptron_tagger')
66
67  text = nltk.word_tokenize("And now for something completely different")
68
69  print(nltk.pos_tag(text))
70
```

Output :

```
PS C:\Users\hp\Desktop\C++> python -u "c:\Users\hp\Desktop\C++\Dynamic Programming\tpp.py"
PS C:\Users\hp\Desktop\C++> python -u "c:\Users\hp\Desktop\C++\Dynamic Programming\tpp.py"
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     C:\Users\hp\AppData\Roaming\nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[('And', 'CC'), ('now', 'RB'), ('for', 'IN'), ('something', 'NN'), ('completely', 'RB'), ('different', 'JJ')]
```

Now understanding method similar , this method takes a word ‘w’ finds all context w1ww2 then finds all words w’ that appear in the same context i.e. w1w’w2 .

Following Code shows example of similar method :

```
nltk.download('brown')

text = nltk.Text(word.lower() for word in nltk.corpus.brown.words())

text.similar('woman')

text.similar('bought')

text.similar('over')

text.similar('the')
```

Output :

```
[nltk_data] Downloading package brown to
[nltk_data]     C:\Users\hp\AppData\Roaming\nltk_data...
[nltk_data]   Package brown is already up-to-date!
man time day year car moment world house family child country boy
state job place way war girl work word
made said done put had seen found given left heard was been brought
set got that took in told felt
in on to of and for with from at by that into as up out down through
is all about
a his this their its her an that our any all one these my in your no
some other and
```

Representing Tagged Tokens :

By convention in NLTK , a tagged token is represented using a tuple consisting of the token and tag . We can create one of those tag using the str2tuple() method

We can construct a list of tagged tokens directly from a string. The first step is to tokenize the string to access the individual word/tag strings, and then to convert each of these into a tuple (using str2tuple()) :

```
###  Representing Tagged Tokens ###

### This way we can tag any word ###
tagged_token = nltk.tag.str2tuple('fly/NN')
print(tagged_token)

### This string is written along with the their tags ###
### and by using str2tuple function we can tag the words
as we have written in the string ###
sent = '''The/AT grand/JJ jury/NN commented/VBD on/IN a/AT
number/NN of/I
other/AP topics/NNS ,/, AMONG/IN them/PPO the/AT Atlanta/
NP and/CC Fulton/NP-tl County/NN-tl purchasing/VBG
departments/NNS which/WDT it/PPS
said/VBD ``/`` ARE/BER well/QL operated/VBN and/CC follow/
VB generally/RB
accepted/VBN practices/NNS which/WDT inure/VB to/IN the/AT
best/JJT
interest/NN of/IN both/ABX governments/NNS ''/'' ./. '''

tagged_word = [nltk.tag.str2tuple(t) for t in sent.split()]

print(tagged_word)
```

Output :

```
('fly', 'NN')
[('The', 'AT'), ('grand', 'JJ'), ('jury', 'NN'), ('commented', 'VBD')
, ('on', 'IN'), ('a', 'AT'), ('number', 'NN'), ('of', 'I'),
('other', 'AP'), ('topics', 'NNS'), (',', ','), ('AMONG', 'IN'), ('th
em', 'PPO'), ('the', 'AT'), ('Atlanta', 'NP'), ('and', 'CC')
, ('Fulton', 'NP-TL'), ('County', 'NN-TL'), ('purchasing', 'VBG'), ('
departments', 'NNS'), ('which', 'WDT'), ('it', 'PPS'), ('sai
d', 'VBD'), ('``', '``'), ('ARE', 'BER'), ('well', 'QL'), ('operated'
, 'VBN'), ('and', 'CC'), ('follow', 'VB'), ('generally', 'RB
'), ('accepted', 'VBN'), ('practices', 'NNS'), ('which', 'WDT'), ('in
ure', 'VB'), ('to', 'IN'), ('the', 'AT')'"), ('.', '.')]
```

Reading Tagged Corpora :

Several of the corpora included with NLTK have been tagged for their part-of-speech. corpora use a variety of formats for storing part-of-speech tags. NLTK's corpus readers provide a uniform interface so that you don't have to be concerned with the different file formats

Note that part-of-speech tags have been converted to uppercase; this has become standard practice since the Brown Corpus was published.

Whenever a corpus contains tagged text, the NLTK corpus interface will have a tagged_words() method

Here are some more examples, again using the output format illustrated for the Brown Corpus:

```
### Reading Tagged Corpora ###

nltk.download('nps_chat')
nltk.download('conll2000')
nltk.download('treebank')

print(nltk.corpus.nps_chat.tagged_words())
print(nltk.corpus.conll2000.tagged_words())
print(nltk.corpus.treebank.tagged_words())
```

Output :

```
[nltk_data] Downloading package nps_chat to
[nltk_data]     C:\Users\hp\AppData\Roaming\nltk_data...
[nltk_data]   Package nps_chat is already up-to-date!
[nltk_data] Downloading package conll2000 to
[nltk_data]     C:\Users\hp\AppData\Roaming\nltk_data...
[nltk_data]   Package conll2000 is already up-to-date!
[nltk_data] Downloading package treebank to
[nltk_data]     C:\Users\hp\AppData\Roaming\nltk_data...
[nltk_data]   Package treebank is already up-to-date!
[('now', 'RB'), ('im', 'PRP'), ('left', 'VBD'), ...]
.
```

A very important feauture to be noted is that Tagged corpora for several other languages are distributed with NLTK, including Chinese, Hindi, Portuguese, Spanish, Dutch, and Catalan. These usually contain non ASCII text, and Python always displays this in hexadecimal when printing a larger structure such as a list.

The above code contains some abbreviation of tags of token , the following table describes all the abbreviation of tag used by the NLTK library :

Tag	Meaning	Examples
ADJ	adjective	*new, good, high, special, big, local*
ADV	adverb	*really, already, still, early, now*
CNJ	conjunction	*and, or, but, if, while, although*
DET	determiner	*the, a, some, most, every, no*
EX	existential	*there, there's*
FW	foreign word	*dolce, ersatz, esprit, quo, maitre*
MOD	modal verb	*will, can, would, may, must, should*
N	noun	*year, home, costs, time, education*
NP	proper noun	*Alison, Africa, April, Washington*
NUM	number	*twenty-four, fourth, 1991, 14:24*
PRO	pronoun	*he, their, her, its, my, I, us*
P	preposition	*on, of, at, with, by, into, under*
TO	the word *to*	*to*
UH	interjection	*ah, bang, ha, whee, hmpf, oops*
V	verb	*is, has, get, do, make, see, run*
VD	past tense	*said, took, told, made, asked*
VG	present participle	*making, going, playing, working*
VN	past participle	*given, taken, begun, sung*
WH	*wh* determiner	*who, which, when, what, where, how*

Unsimplified Tags :

The following code finds all tags starting with NN, and provides a few example words for each one . We will see that there are many variants of NN; the most important contain $ for possessive nouns, S for plural nouns (since plural nouns typically end in s), and P for proper nouns. In addition, most of the tags have suffix modifiers: -NC for citations, -HL for words in headlines, and -TL for titles (a feature of Brown tags)

```python
# Unsimplified Tags

def findtags(tag_prefix, tagged_text):
    cfd = nltk.ConditionalFreqDist((tag, word) for (word, tag) in tagged_text if tag.startswith(tag_prefix))
    return {tag: list(cfd[tag].keys())[:5] for tag in cfd.conditions()}

nltk.download('brown')
nltk.download('universal_tagset')

# Get the tagged words from the Brown corpus (news category)
tagged_words = nltk.corpus.brown.tagged_words(categories='news', tagset='universal')

# Find tags with the prefix 'NN' (noun-related tags)
tagdict = findtags('NOUN', tagged_words)

# Print the results
for tag in sorted(tagdict):
    print(tag, tagdict[tag])
```

Output :

```
[nltk_data] Downloading package brown to
[nltk_data]     C:\Users\hp\AppData\Roaming\nltk_data...
[nltk_data]   Package brown is already up-to-date!
[nltk_data] Downloading package universal_tagset to
[nltk_data]     C:\Users\hp\AppData\Roaming\nltk_data...
[nltk_data]   Package universal_tagset is already up-to-date!
NOUN ['Fulton', 'County', 'Jury', 'Friday', 'investigation']
```

Learning to explore and process text data with pandas and extract features :

To explore and process the data woth help of pandas we neeed to first load the data then clean the data and then extract it's feautures .

First , install pandas by typing the command "pip install pandas".

Second , load the text , for example a csv file can be loaded as " data = pandas.read_csv('fileInput.csv') "

Now the following code display some Basic exploration of the data :

```
# Displays the first few rows of the data input file
print(df.head())

# Summarises the statistics of the data
print(df.describe())

# Information about data types
print(df.info())
```

Third , Now the data should be cleaned before analysis , common cleaning steps include :

1 . Converting text to lowercase

2 . removing punctuation

3 . Removing stopwords

4 . Tokenization (splitting into words)

Following code demonstrates the process of cleaning the data :

```
import pandas as pd
import nltk
import string
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer

nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')

df = pd.load_csv('file.csv')

# Sample text
df['text'] = df['text'].astype(str)

# first check that is Convert to lowercase
df['cleaned_text'] = df['text'].str.lower()

# second check that is Remove punctuation
df['cleaned_text'] = df['cleaned_text'].str.replace('[{}]'.
format(string.punctuation), '')

# third check that is Remove stopwords
stop_words = set(stopwords.words('english'))
df['cleaned_text'] = df['cleaned_text'].apply(lambda x: '
'.join([word for word in word_tokenize(x) if word not in
stop_words]))
```

After cleaning the data we need to extract feauture

Feature extraction involves transforming text data into numerical features that can be used in machine learning models. Common techniques include:

- **Bag of Words (BoW)**
- **Term Frequency-Inverse Document Frequency (TF-IDF)**

- Word Embeddings (Word2Vec, GloVe, etc.)

Word Embeddings (Word2Vec) :

So The first question which arises is what is word embedding ?

- ➔ Word Embedding is a language modelling technique for mapping words to vectors of real numbers . It represents words or phrases in vector space with several dimensions . Word Embedding can be generated using various methods like neural networks , probabilistic methods.
- ➔ **Word2Vec** consists Of models for generating word embedding , these models are shallow two layers neural networks having one input layers , one hidden layer and one output layer .

Next things that arises in our mind is what is Word2Vec(In detail) :

- ➔ Word2Vec is a method used in NLP to map words/phrases to a vector in a vector space .It is an effort to map words to high – dimensional vector , such that it caputers the semantic relationship between words , also it is designed such that the similar words are mapped to similar vector representations .
- ➔ Two Get a more deeper knowledge of Word2Vec we need to understand the two architectures which it utilizes :
  - **CWOB(Continous bag of words)** :
    The CWOB model predicts the current word given the context and it uses a hidden a layer contains the dimensions we want to represent the current word present at the output layer . The output layer contains the output word , input layer contains the context words .
  - Skip Gram :

    Skip gram predicts context words given the current words(input)

, The hidden layer contain the number of dimensions in which we want to represent the current word at the input layer .

The crux of word embedding is that words that occur in similar context tend to closer to each other in vector space .

The modules which we need in PYTHON for generating words to vector are "nltk" and "genism" .

Now before we began in python that how to implement it first let's see the advantages of using Word2Vec :

1. **Sematic Representations :** Word2Vec records the connection between words semantically . Words are represented in the vector space so that similar words are near to one another . This enables the model to interpret the word according to their context .
2. **Vector Arithemetic :** word2Vec stores the words such that they have algebraic characteristics . Following is it's example – **"France" + "Paris" – "Delhi" = "India"**
3. **Efficiency :** Word2Vec's high efficiency enabels to use it for big dataset , learning high – dimensional vectors with high vocalbulary requires high efficiency .
4. **Wide range of applications :** Word2Vec embedding is implemented in a number of NLP applications , machine translation , text classification , text classification , sentiment analysis and information retrtival .
5. **Scalibilty :** It can handle big copora with very ease and is scalable , which is essential for training of large datasets .
6. **Open source implementations :** Word2Vec has open – source versions , including one that is included in the Genism library , It's widespread adpotation and use in research and industry can be attributed in part to its accessibility .

Implementing Word2Vec using Python :

```python
from gensim.models import Word2Vec
import gensim
from nltk.tokenize import sent_tokenize
from nltk.tokenize import word_tokenize
import warnings

warnings.filterwarnings(action = "ignore")

sample = open("./data.txt")
s1 = sample.read()
s = s1.replace("\n" , " ")

DATA = []

for i in sent_tokenize(s) :

    temp = []

    for j in range word_tokenize(i) :
        temp.append(j.lower())

    DATA.append(temp)


model1 = gensim.models.Word2Vec(DATA,min_count =1 , vector_size = 100 , window = 5)


print("Cosine similarity between 'king' " + "and 'man' - CBOW : ",model1.wv.similarity('king', 'man'))
print("Cosine similarity between 'king' " +"and 'woman' - CBOW : ",model1.wv.similarity('king', 'woman'))

#Create Skip Gram model
model2 = gensim.models.Word2Vec(DATA, min_count=1, vector_size=100,window=5, sg=1)

print("Cosine similarity between 'king' " +"and 'man' - Skip Gram : ", model2.wv.similarity('king', 'man'))
print("Cosine similarity between 'king' " +"and 'woman' - Skip Gram : ",model2.wv.similarity('king', 'woman'))
```

The above is the implementation of Word2Vec and printing the similarity between "King" and "Man" , "king" and "woman" .

The ouput of the following is code is as follows :

Cosine similarity between 'king' and 'man' - CBOW : 0.999249298413

Cosine similarity between 'king' and 'woman' - CBOW : 0.974911910445

Cosine similarity between 'king' and 'man' - Skip Gram : 0.885471373104

Cosine similarity between 'king' and 'woman' - Skip Gram : 0.856892599521

RNNs and LSTM :

Before going onto RNN And LSTM I would to brief about the Neural Networks and explaining few of the terminologies used below –

Neural Networks (NNs) are a foundational concept in machine learning, inspired by the structure and function of the human brain. At their core, NNs consist of interconnected nodes organized into layers. Input layers receive data, hidden layers process information, and output layers produce results. The strength of NNs lies in their ability to learn from data, adjusting internal parameters (weights) during training to optimize performance.

Forward propogation – In the **forward propagation** phase, data travels through the network, and computations occur at each layer, generating predictions. It's similar to information flowing from input to output.

Backward propogation : The **backward propagation** phase involves the crucial aspect of learning. Through techniques like gradient descent, the network refines its internal parameters by calculating the gradient of the loss function with respect to the weights. The chain rule plays a pivotal role here, allowing the network to attribute the loss to specific weights, enabling fine-tuning for better accuracy.

Chain Rule :The chain rule in calculus is the linchpin for backpropagation. It enables the computation of partial derivatives, attributing the network's overall error to individual weights. This decomposition is crucial for making nuanced adjustments during training.

Gradient Descent : Gradient descent is the driving force behind weight adjustments in NNs. It's an optimization algorithm that minimizes the

loss function by iteratively moving toward the steepest downhill direction in the multidimensional weight space. This iterative adjustment of weights enhances the network's predictive accuracy. Important Point is that Gradient Descend can be used when there is only minima or maxima .

Recurrent Neural Networks (RNN) : RNNs are a specialized form of NN designed to handle sequential data. They introduce the concept of memory, enabling the network to retain information about previous inputs. This memory is crucial for tasks where context matters, such as language understanding and generation.

- **Sequential Processing** : Unlike traditional neural networks , RNNs are the one designed to process the sequences of data, by taking input input in sequential manner , It is very important to do so because in language sequence / order the words occur matter's a lot .
- **Recurrent Connections** : It is a key feature of RNN , because it allows the network to retain a form of "memory". At each step in a sequence , the RNN process current input along with a "hidden state" from the previous step . This hidden state contains information learned from prior inputs .
- **Hidden State :** It is updated each step , based on new inputs and the previous hidden state . This allows RNN to carry Info across different steps in the sequence .
- **Shared Weights** : In a RNN , the parameters are shares across all time steps . This means the same weight are used to process each input in the sequence , making the model more efficient and reducing the numbers of parameters .

- **Use Cases Of RNNs** : RNNs find applications in Natural Language Processing (language modeling, machine translation), Speech Recognition (phoneme recognition, voice synthesis), and Time Series Prediction (stock price forecasting, weather prediction).

Long Short-Term Memory (LSTM) :

LSTMs is a advanced type of RNN , specifally enginerred to address and overcome the limitations inherent in traditional RNNs , particulary when dealing with long – term dependencies .

How It Works :

- **Advanced Memory Handling:** The defining feature of LSTM is its complex memory cell, known as the LSTM unit. This unit can maintain information over extended periods, thanks to its unique structure comprising different gates.

- **Gating Mechanism:** LSTMs incorporate three types of gates, each playing a crucial role in the network's memory management.

- **Input Gate:** Determines which values from the input should be used to modify the memory.

- **Forget Gate:** Decides what portions of the existing memory should be discarded.

- **Output Gate:** Controls the output flow of the memory content to the next layer in the network.

- **Cell State**: It allows information to flow relatively unchanged and ensures that the network retains and accesses important long-term information efficiently.

Use Cases :

LSTMs have proven effective in various domains that require the processing of sequences with long-term dependencies, such as complex sentence structures in text, speech recognition, and time-series analysis.

When to Use Which One :

RNN : Ideal for processing sequences and maintaining information over short time spans. Simple architecture makes them computationally efficient.

LSTM : Highly effective in learning long-term dependencies. The addition of input, forget, and output gates allows for better control over the memory cell, making them adept at handling issues like the vanishing gradient problem.

I haven't practiced RNN and LSTM in code because they are pretty difficult to code and uses a lot modules which we aren't familiar with (like tensor flow) that part , So concluding the RNNs and LSTM only with the theoretical part.

Learning About Transformers And BERT :

Transformers :

It is A Neural Network architecture used for ML tasks , the concept of which was introduced by "Vaswani et al " in a research paper " **Attention is All you need** "

The RNN model suffers from **'gradient vanishing problem'** which causes long term memory loss .

Example of what's the actual problem of RNN and why need Transformers as an alternative of it : Consider the following sentence "Haresh went to 'Uttarakhand' this summer and he was overwhelmed by the beauty of nature ." Now , if we ask who does "he" refers , It couldn't be able to recall that he is 'haresh' . The sequential nature of processing means that the model was trained at word level , not the sentence level . The gradient carry info used in RNN parameter update and when the gradient becomes smaller , then no real learning is done .

Working And of transformers :

- Positional Encoding : Tranformers not interpret data in a sequential order of elements , rather they use positional encoding into the input embedding , which serve s to convey information about the specific positions of tokens within the sequence .

- Feedforward Network of Transformers : The feedforward network consists if a fully connected layer for followed by non-linear activation function, such as Rectified Linear Unit (ReLU). The model

captures and process features at different position in the sequence. These networks in both the encoder and decoder operate independently on each position.

- Attention mechanism : This mechanism employs a scalar dot – product approach , where the computation involves scaled dot product between the query , key and value vectors . This produces weighted Values that are then summed to yield the attention output. To enhance the models understanding of sematincs of words , a multi head attention mechanism is incorporated , which involves applying the attention model multiple times concurrently, each with distinct learned linear projections of the input. The resulting outputs from these parallel computations are concatenated and undergo a linear transformation to generate the final attention result.

Uses Of Transformers :

- It is used in NLP like tasks , text summarization , sentiment analysis etc .
- Other application of Transformers include "Speech recognition" , "Computer Vision (such as image classification) " , object detection image generation etc .

BERT :

BERT stands for Bidirectional Encoder Representation from Transformers , it is an open ML framework designed specifically for NLP task . This framework was created at Google .

BERT leverages a transformer based NN (Neural Network) to understand and generate human – like language . BERT employs an encoder – only architecture . In the original Architecure there are both encoder and decoder modules , So the idea behind using only encoder in BERT is to emphasis on understanding input sequences rather than generating output sequences .

Bidirectional Approach Of BERT :

Traditional language model process data sequentially that is either L to R OR R to L , This method limts the model awarness to the immediate context precciding the target word , To oovercome this BERT looks at all the words in a sentence sequentially .

For example consider this :

"The bank is situated on the _____ of the river" .

A normal Text processing model such as RNN would consider only the part "The bank is situated on the" to would find difficult to understand whether the "bank" refers to a "financial institution" or a "side of river" .

Whereas BERT would simultaneously understand the right and the left context enabling a much better understanding .

And Important point is that architecture and working principle is a hard topic to understand an I'm skipping this part as it requires some pre requisites and moving onto implementing Bert .

Implementing BERT :

1. First Run the command "pip install transformers" to install the module "transformers"

2. Below is the python code for it's implementation :

```python
from transformers import BertTokenizer

# Load pre-trained BERT tokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-cased")

# Input text
text = 'ChatGPT is a language model developed by OpenAI, based on the GPT (Generative Pre-trained Transformer) architecture. '

# Tokenize and encode the text
encoding = tokenizer.encode(text)

# Print the token IDs
print("Token IDs:", encoding)

# Convert token IDs back to tokens
tokens = tokenizer.convert_ids_to_tokens(encoding)

# Print the corresponding tokens
print("Tokens:", tokens)
```

The Output of the above is as follows :

```
PS C:\Users\hp\Desktop\w5 to w8> python -u "c:\Users\hp\Desktop\w5 to w8\temp.py"
None of PyTorch, TensorFlow >= 2.0, or Flax have been found. Models won't be available and only tokenizers,
Token IDs: [101, 24705, 1204, 17095, 1942, 1110, 170, 1846, 2235, 1872, 1118, 3353, 1592, 2240, 117, 1359,
19, 102]
Tokens: ['[CLS]', 'Cha', '##t', '##GP', '##T', 'is', 'a', 'language', 'model', 'developed', 'by', 'Open', '
', 'trained', 'Trans', '##former', ')', 'architecture', '.', '[SEP]']
```

With the BERT being last topic of my POA , I conclude my journey and I was Overwhelmed to pursue this topic in my SOS as it added a lot in my skill set , I would like to thanks the MnP Club for assigning me such a good mentor . "SIGNING OFF "