Index

Experiment 1

Word Analysis

AIM: To learn about morphological features of a word by analyzing it.

Problem Statement: A word can be simple or complex. For example, the word 'cat' is simple because one cannot further decompose the word into smaller part. On the other hand, the word 'cats' is complex, because the word is made up of two parts: root 'cat' and plural suffix '-s'.

Description: Analysis of a word into root and affix(es) is called as Morphological analysis of a word. It is mandatory to identify root of a word for any natural language processing task. A root word can have various forms. For example, the word 'play' in English has the following forms: 'play', 'plays', 'played' and 'playing'. Hindi shows more number of forms for the word 'खेल' (khela) which is equivalent to 'play'. The forms of 'खेल'(khela) are the following:
खेल(khela), खेला(khelaa), खेली(khelii), खेलूंगा(kheluungaa), खेलूंगी(kheluungii), खेलेगा(khelegaa), खेलेगी(khelegii), खेलते(khelate), खेलती(khelatii), खेलने(khelane), खेलकर(khelakar)

Types of Morphology: Morphology is of two types,

Inflectional morphology :- Deals with word forms of a root, where there is no change in lexical category. For example, 'played' is an inflection of the root word 'play'. Here, both 'played' and 'play' are verbs.

Derivational morphology :- Deals with word forms of a root, where there is a change in the lexical category. For example, the word form 'happiness' is a derivation of the word 'happy'. Here, 'happiness' is a derived noun form of the adjective 'happy'.

Morphological Features: All words will have their lexical category attested during morphological analysis. A noun and pronoun can take suffixes of the following features: gender, number, person, case.

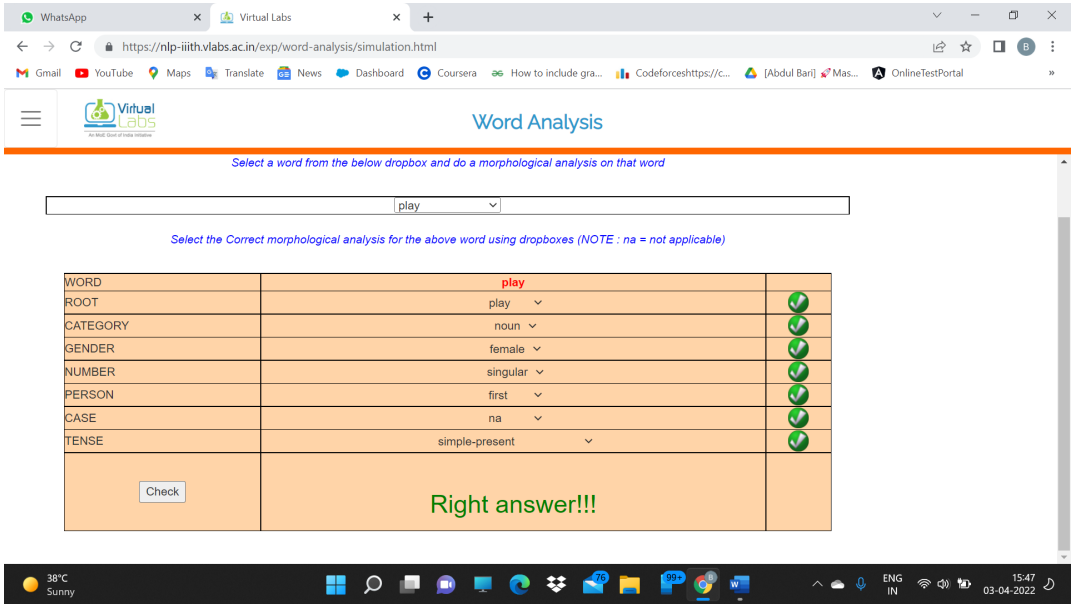
The value of tense can be present, past or future. This feature is applicable for verbs. This feature is not applicable for verbs.'case' can be direct or oblique. This feature is applicable for nouns. A case is an oblique case when a postposition occurs after noun. If no postposition can occur after noun, then the case is a direct case.

Procedure:

```
STEP 1: Select the language.
```

```
OUTPUT: Drop down for selecting words will appear.

STEP 2: Select the word.

OUTPUT: Drop down for selecting features will appear.

STEP 3: Select the features.

STEP 4: Click "Check" button to check your answer.

OUTPUT: Right features are marked by tick and wrong features are marked by
cross.
```

Result:



Conclusion: In the above experiment, we have morphologically analysed the given word.

Experiment 2

Word Generation

AIM: To generate word forms from root and suffix information.

Problem Statement: A word can be simple or complex. For example, the word 'cat' is simple because one cannot further decompose the word into smaller part. On the other hand, the word 'cats' is complex, because the word is made up of two parts: root 'cat' and plural suffix '-s'

Description: Given the root and suffix information, a word can be generated. For example,

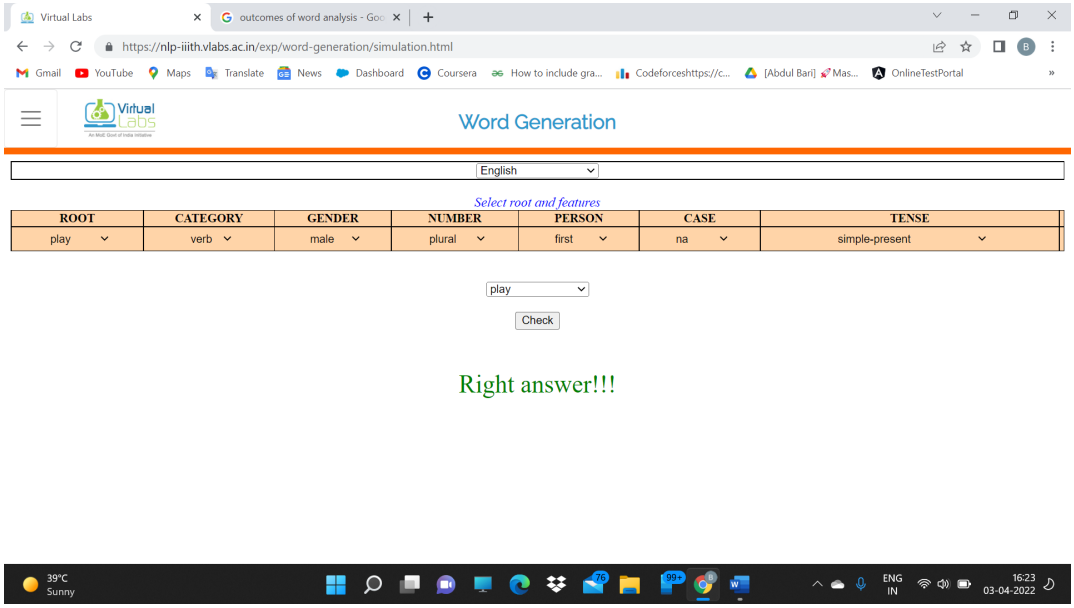
Language	input:analysis	output:word
Hindi	rt=लड़का(ladakaa), cat=n, gen=m, num=sg, case=obl	लड़के(ladake)
Hindi	rt=लड़का(ladakaa), cat=n, gen=m, num=pl, case=dir	लड़के(ladake)
English	rt=boy, cat=n, num=pl	boys
English	rt=play, cat=v, num=sg, per=3, tense=pr	plays

Morphological analysis and generation: Inverse processes. Analysis may involve non-determinism, since more than one analysis is possible. Generation is a deterministic process. In case a language allows spelling variation, then till that extent, generation would also involve non-determinism.

Procedure:

```
STEP 1: Select the language.

OUTPUT: Drop downs for selecting root and other features will appear.

STEP 2: Select the root and other features.

STEP 3: After selecting all the features, select the word corresponding above
features selected.

STEP 4: Click the check button to see whether right word is selected or not

OUTPUT: Output tells whether the word selected is right or wrong
```

Result:



Conclusion: In the above experiment, we have generated the forms of word from root and suffix information.

Experiment 3

Morphology

AIM: Understanding 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 i.e., morphemes. A morpheme is the smallest meaningful linguistic unit.

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".

बच्चा	-ओं
लड़का	-ओं
play	-ed
want	-ed

Words can be analysed morphologically if we know all variants of a given root word. We can use an 'Add-Delete' table for this analysis.

Description: 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.

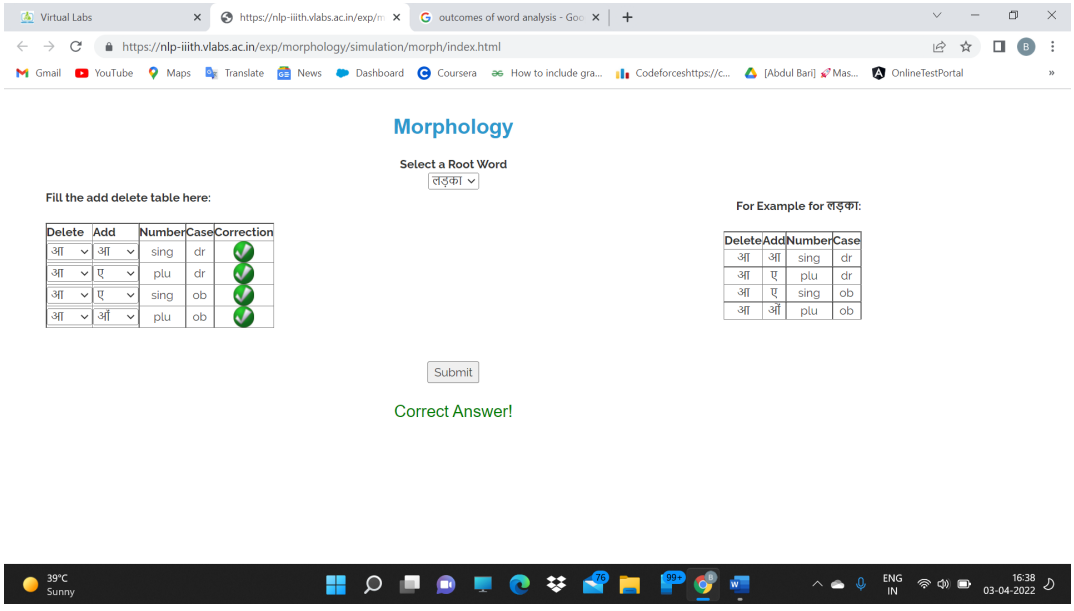
Add-Delete table for बच्चा

Delete	Add	Number	Case	Variants
Aa	aa	sing	dr	bachchaa
Aa	e	Plu	dr	bachche
Aa	e	Sing	ob	bachche
aa	oM	Plu	ob	bachchoM

Procedure:

```
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.
```

Result:



Conclusion: In the above experiment, we have understood the morphology of a word by using add-delete tables.

Experiment 4

N-Grams

AIM: To learn how to calculate bigrams from a given corpus and calculate probability of a sentence.

Problem Statement: Probability of a sentence can be calculated by the probability of sequence of words occuring in it. We can use Markov assumption, that the probability of a word in a sentence depends on the probability of the word occuring just before it. Such a model is called first order Markov model or the bigram model.

$$P(W_n \mid W_{n-1}) = P(W_{n-1},W_n)/P(W_{n-1})$$

Here, Wn refers to the word token corresponding to the nth word in a sequence.

Description: A combination of words forms a sentence. However, such a formation is meaningful only when the words are arranged in some order. Eg: Sit I car in the

Such a sentence is not grammatically acceptable. However some perfectly grammatical sentences can be nonsensical too! Eg: Colorless green ideas sleep furiously

One easy way to handle such unacceptable sentences is by assigning probabilities to the strings of words i.e, how likely the sentence is.

Probability of a sentence If we consider each word occurring in its correct location as an independent event,the probability of the sentences is : P(w(1), w(2)..., w(n-1), w(n)) Using chain rule:

$$= P(w(1)) * P(w(2)|w(1)) * P(w(3)|w(1)w(2)) \ldots P(w(n)|w(1)w(2) \ldots w(n-1))$$

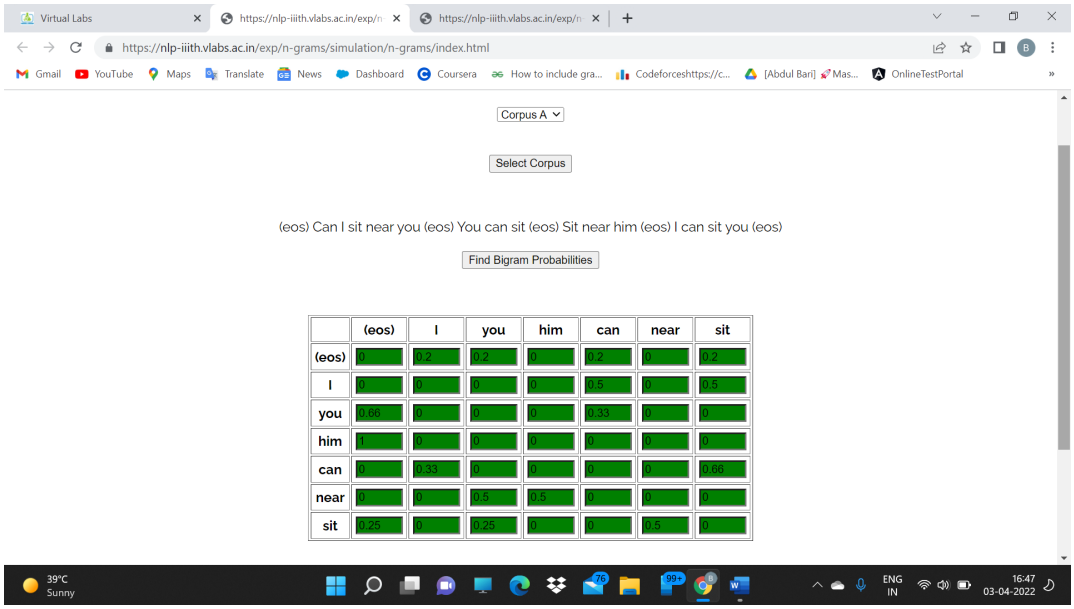
Bigrams :- We can avoid this very long calculation by approximating that the probability of a given word depends only on the probability of its previous words. This assumption is called Markov assumption and such a model is called Markov model- bigrams. Bigrams can be generalized to the n-gram which looks at (n-1) words in the past. A bigram is a first-order Markov model. Therefore , P(w(1), w(2)..., w(n-1), w(n)) = P(w(2)|w(1)) P(w(3)|w(2)) ... P(w(n)|w(n-1)) We use (eos) tag to mark the beginning and end of a sentence.

A bigram table for a given corpus can be generated and used as a lookup table for calculating probability of sentences. Eg: Corpus - (eos) You book a flight (eos) I read a book (eos) You read (eos)

Procedure:

```
STEP 1: Select a corpus and click on Generate bigram table

STEP 2: Fill up the table that is generated and hit Submit

STEP 3: If incorrect (red), see the correct answer by clicking on show answer
or repeat Step 2.

STEP 4: If correct (green), click on take a quiz and fill the correct answer
```

Result:



	(eos)	I	you	him	can	near	sit
(eos)	0	0.2	0.2	0	0.2	0	0.2
I	0	0	0	0	0.5	0	0.5
you	0.66	0	0	0	0.33	0	0
him	1	0	0	0	0	0	0
can	0	0.33	0	0	0	0	0.66
near	0	0	0.5	0.5	0	0	0
sit	0.25	0	0.25	0	0	0.5	0

Conclusion: In this experiment , we have learnt to calculate bigrams and probability of a sentence.

Experiment 5

N-Grams Smoothing

AIM: To learn how to apply add-one smoothing on sparse bigram table.

Problem Statement: One major problem with standard N-gram models is that they must be trained from some corpus, and because any particular training corpus is finite, some perfectly acceptable N-grams are bound to be missing from it. We can see that bigram matrix for any given training corpus is sparse. There are large number of cases with zero probabilty bigrams and that should really have some non-zero probability. This method tend to underestimate the probability of strings that happen not to have occurred nearby in their training corpus.

There are some techniques that can be used for assigning a non-zero probabilty to these 'zero probability bigrams'. This task of reevaluating some of the zero-probability and low-probabilty N-grams, and assigning them non-zero values, is called smoothing.

	eos	I	booked	a	flight	took
eos	0	300	0	0	0	300
I	0	0	300	0	0	0
booked	0	0	0	300	0	0
a	0	0	0	0	600	0
flight	600	0	0	0	0	0
took	0	0	0	300	0	0

Valid bigrams absent in the training corpus:
How could I eos I have a booked room eos I took a flight eos

Description: The standard N-gram models are trained from some corpus. The finiteness of the training corpus leads to the absence of some perfectly acceptable N-grams. This results in sparse bigram matrices. This method tend to underestimate the probability of strings that do not occur in their training corpus.

There are some techniques that can be used for assigning a non-zero probabilty to these 'zero probability bigrams'. This task of reevaluating some of the zero-probability and low-probabilty N-grams, and assigning them non-zero values, is called smoothing.

Add-One Smoothing :- In Add-One smooting, we add one to all the bigram counts before normalizing them into probabilities. This is called add-one smoothing.

Application on unigrams :- The unsmoothed maximum likelihood estimate of the unigram probability can be computed by dividing the count of the word by the total number of word tokens N. P(wx) = c(wx)/sumi{c(wi)} = c(wx)/N Let there be an adjusted count c. ci = (c i+1 *N/(N+V)), where V is the total number of word types in the language. Now, probabilities can be calculated by normalizing counts by N. pi* = (c i+1)/(N+V)

Application on bigrams :- Normal bigram probabilities are computed by normalizing each row of counts by the unigram count: P(wn|wn-1) = C(wn-1wn)/C(wn-1) For add-one smoothed bigram counts we need to augment the unigram count by the number of total word types in the vocabulary V: p*(wn|wn-1) = (C(wn-1wn)+1)/(C(wn-1)+V)

Procedure:

```
STEP 1: Select a corpus

STEP 2: Apply add one smoothing and calculate bigram probabilities using the
given bigram counts,N and V. Fill the table and hit Submit

STEP 3: If incorrect (red), see the correct answer by clicking on show answer
or repeat Step 2
```

Result:

Conclusion: In this experiment ,we have learnt to apply add-on smoothing on sparse bigram.

Experiment 6

Building POS Tagger

AIM: To know the importance of context and size of training corpus in learning parts of speech.

Problem Statement: In corpus linguistics, part-of-speech tagging (POS tagging or POST), also called grammatical tagging or word-category disambiguation, is the process of marking up a word in a text (corpus) as corresponding to a particular part of speech, based on both its definition, as well as its context i.e. relationship with adjacent and related words in a phrase, sentence, or paragraph. A simplified form of this is identification of words as nouns, verbs, adjectives, adverbs, etc. Once performed by hand, POS tagging is now done in the context of computational linguistics, using algorithms which associate discrete terms, as well as hidden parts of speech, in accordance with a set of descriptive tags. POS-tagging algorithms fall into two distinctive groups: rule-based and stochastic.

Description: Hidden Markov Model :- In the mid 1980s, researchers in Europe began to use Hidden Markov models (HMMs) to disambiguate parts of speech. HMMs involve counting cases, and making a table of the probabilities of certain sequences. For example, once you've seen an article such as 'the', perhaps the next word is a noun 40% of the time, an adjective 40%, and a number 20%. Knowing this, a program can decide that "can" in "the can" is far more likely to be a noun than a verb or a modal. The same method can of course be used to benefit from knowledge about following words.

More advanced ("higher order") HMMs learn the probabilities not only of pairs, but triples or even larger sequences. So, for example, if you've just seen an article and a verb, the next item may be very likely a preposition, article, or noun, but much less likely another verb.

It is worth remembering, as Eugene Charniak points out in Statistical techniques for natural language parsing, that merely assigning the most common tag to each known word and the tag "proper noun" to all unknowns, will approach 90% accuracy because many words are unambiguous. HMMs underlie the functioning of stochastic taggers and are used in various algorithms. Accuracies for one such algorithm (TnT) on various training data is shown here.

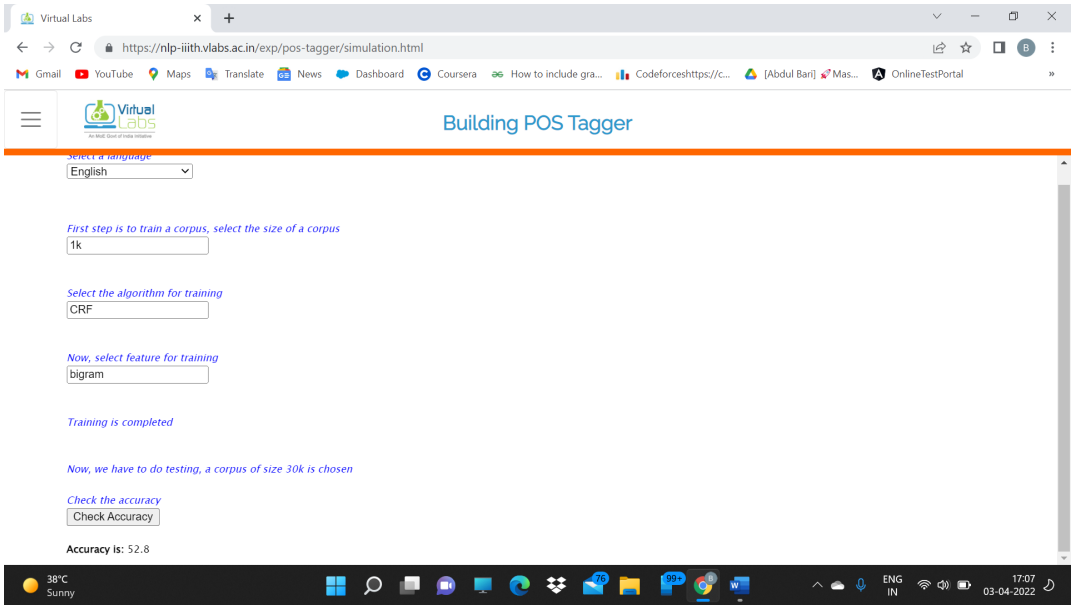
Conditional Random Field :- Conditional random fields (CRFs) are a class of statistical modelling method often applied in machine learning, where they are used for structured prediction. Whereas an ordinary classifier predicts a label for a single sample without regard to "neighboring" samples, a CRF can take context into account. Since it can consider context, therefore CRF can be used in Natural Language Processing. Hence, Parts of Speech tagging is also possible. It predicts the POS using the lexicons as the context. If only one neighbour is

considered as a context, then it is called bigram. Similarly, two neighbours as the context is called trigram. In this experiment, size of training corpus and context were varied to know their importance.

Procedure:

```
STEP1: Select the corpus.

STEP2: For the given corpus fill the emission and transition matrix. Answers
are rounded to 2 decimal digits.

STEP3: Press Check to check your answer.

Wrong answers are indicated by the red cell.
```

Result:



Conclusion: In the above experiment, we have learnt the importance of context and size of training corpus.

Experiment 7

Chunking

AIM: To understand the concept of chunking and get familiar with the basic chunk tagset.

Problem Statement: Chunking of text invloves dividing a text into syntactically correlated words. For example, the sentence 'He ate an apple.' can be divided as follows:

[NP He] [VP ate] [NP an apple]

Each chunk has an open boundary and close boundary that delimit the word groups as a minimal non-recursive unit. This can be formally expressed by using IOB prefixes.

[NP He] [VP ate] [NP an apple]

Description: Chunking of text involves :- dividing a text into syntactically correlated words. Eg: He ate an apple to satiate his hunger. [NP He] [VP ate] [NP an apple] [VP to satiate] [NP his hunger]

Chunk Types :- The chunk types are based on the syntactic category part. Besides the head a chunk also contains modifiers (like determiners, adjectives, postpositions in NPs).

The basic types of chunks in English are:

	Chunk Type	Tag Name
1	Noun	NP
2	Verb	VP
3	Adverb	ADVP
4	Adjective	ADJP
5	Prepositional	PP

The basic Chunk Tag Set for Indian Languages

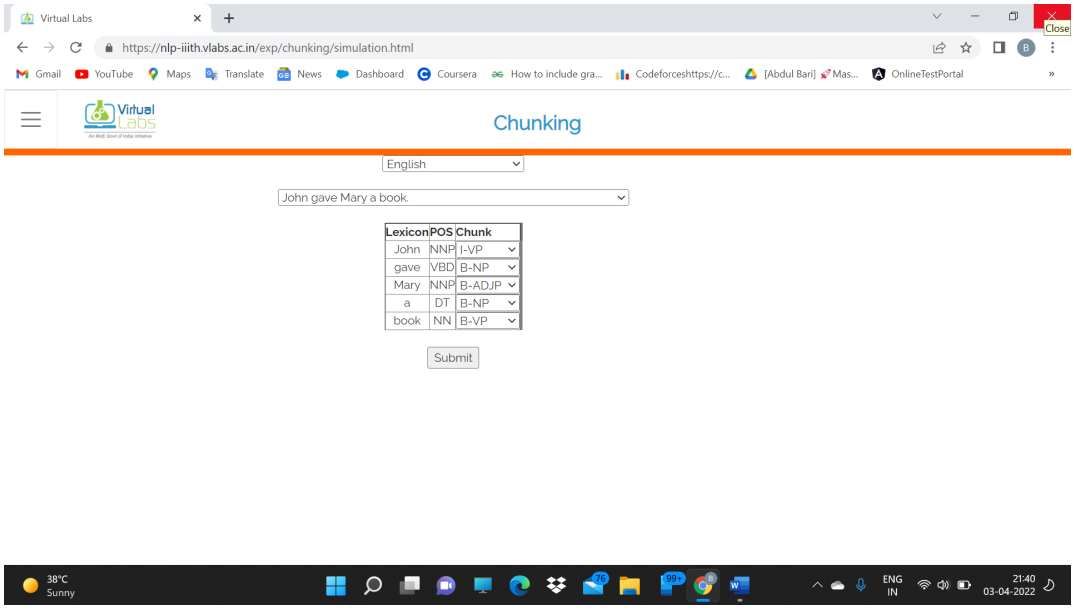
S.No	Chunk Type	Tag Name
1	Noun Chunk	NP

S.No	Chunk Type	Tag Name
2	Finite Verb Chunk	VGF
3	Non-Finite Verb Chunk	VGNF
4	Adjectivial Chunk	JJP
5	Adverb Chunk	RBP

Procedure:

```
STEP1: Select a language

STEP2: Select a sentence

STEP3: Select the corresponding chunk-tag for each word in the sentence and
click the Submit button.

OUTPUT1: The submitted answer will be checked.

Click on Get Answer button for the correct answer.
```

Result:



Conclusion: In the above experiment, we have understood the concept of chunking .

Experiment 8

Building Chunker

AIM: To know the importance of selecting proper features for training a model and size of training corpus in learning how to do chunking.

Problem Statement: Chunking is an analysis of a sentence which identifies the constituents (noun groups, verbs, verb groups, etc.) which are correlated. These are non-overlapping regions of text. Usually, each chunk contains a head, with the possible addition of some function words and modifiers either before or after depending on languages. These are non-recursive in nature i.e. a chunk cannot contain another chunk of the same category. Some of the groups possible are: Noun Group , Verb Group. For example, the sentence 'He reckons the current account deficit will narrow to only 1.8 billion in September.' can be divided as follows:

Each chunk has an open boundary and close boundary that delimit the word groups as a minimal non-recursive unit.

Description: Hidden Markov Model :- In the mid 1980s, researchers in Europe began to use Hidden Markov models (HMMs) to disambiguate parts of speech. HMMs involve counting cases, and making a table of the probabilities of certain sequences. For example, once you've seen an article such as 'the', perhaps the next word is a noun 40% of the time, an adjective 40%, and a number 20%. Knowing this, a program can decide that "can" in "the can" is far more likely to be a noun than a verb or a modal. The same method can of course be used to benefit from knowledge about following words.

When several ambiguous words occur together, the possibilities multiply. However, it is easy to enumerate every combination and to assign a relative probability to each one, by multiplying together the probabilities of each choice in turn.

It is worth remembering, as Eugene Charniak points out in Statistical techniques for natural language parsing, that merely assigning the most common tag to each known word and the tag "proper noun" to all unknowns, will approach 90% accuracy because many words are unambiguous.

HMMs underlie the functioning of stochastic taggers and are used in various algorithms. Accuracies for one such algorithm (TnT) on various training data is shown here.

Conditional Random Field-- Conditional random fields (CRFs) are a class of statistical modelling method often applied in machine learning, where they are used for structured prediction. Whereas an ordinary classifier predicts a label for a single sample without regard to

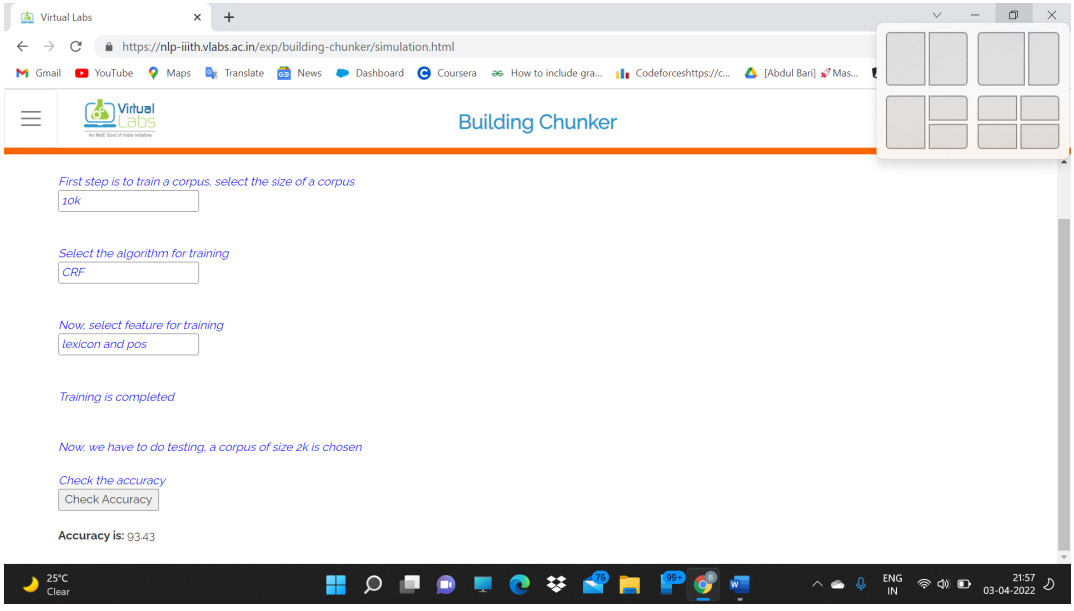
"neighboring" samples, a CRF can take context into account. Since it can consider context, therefore CRF can be used in Natural Language Processing. Hence, Parts of Speech tagging is also possible. It predicts the POS using the lexicons as the context.

In this experiment both algorithms are used for training and testing data. As the size of training corpus increases, it is observed that accuracy increases. Further, even features also play an important role for better output. In this experiment, we can see that Parts of Speech as a feature performs better than only lexicon as the feature. Therefore, it is important to select proper features for training a model to have better accuracy.

Procedure:

```
STEP1: Select the language.

OUTPUT: Drop down to select size of corpus, algorithm and features will
appear.

STEP2: Select corpus size.

STEP3: Select algorithm "CRF" or "HMM".

STEP4: Select feature "only lexicon", "only POS", "lexicon and POS".

OUTPUT: Corresponding accuracy wil be shown.
```

Result:



Conclusion:In the above experiment, we have learnt the importance of selecting proper features for model training.

Experiment 9

Python Library

AIM: To study Natural language processing library in python: NLTK/ any other.

Description: The Natural Language Toolkit, or more commonly NLTK, is a suite of libraries and programs for symbolic and statistical natural language processing (NLP) for English written in the Python programming language. It was developed by Steven Bird and Edward Loper in the Department of Computer and Information Science at the University of Pennsylvania. NLTK includes graphical demonstrations and sample data. It is accompanied by a book that explains the underlying concepts behind the language processing tasks supported by the toolkit, plus a cookbook.

NLTK is a leading platform for building Python programs to work with human language data. It provides easy-to-use interfaces to over 50 corpora and lexical resources such as WordNet, along with a suite of text processing libraries for classification, tokenization, stemming, tagging, parsing, and semantic reasoning, wrappers for industrial-strength NLP libraries, and an active discussion forum.

NLTK is intended to support research and teaching in NLP or closely related areas, including empirical linguistics, cognitive science, artificial intelligence, information retrieval, and machine learning. NLTK has been used successfully as a teaching tool, as an individual study tool, and as a platform for prototyping and building research systems. There are 32 universities in the US and 25 countries using NLTK in their courses. NLTK supports classification, tokenization, stemming, tagging, parsing, and semantic reasoning functionalities.

Thanks to a hands-on guide introducing programming fundamentals alongside topics in computational linguistics, plus comprehensive API documentation, NLTK is suitable for linguists, engineers, students, educators, researchers, and industry users alike. NLTK is available for Windows, Mac OS X, and Linux. Best of all, NLTK is a free, open source, community-driven project.

NLTK has been called “a wonderful tool for teaching, and working in, computational linguistics using Python,” and “an amazing library to play with natural language.”

Things you can do with NLTK —

Tokenization: The breaking down of text into smaller units is called tokens. tokens are a small part of that text. If we have a sentence, the idea is to separate each word and build a

vocabulary such that we can represent all words uniquely in a list. Numbers, words, etc.. all fall under tokens.

Input:

```
from nltk.tokenize import sent_tokenize, word_tokenize
text = "Natural language processing is an exciting area."
print(sent_tokenize(text))
```

Output:

```
'Natural language processing is an exciting area.', 'Huge budget have been
allocated for this.'
```

Lower case conversion: We want our model to not get confused by seeing the same word with different cases like one starting with capital and one without and interpret both differently. So we convert all words into the lower case to avoid redundancy in the token list.

Input:

```
text = re.sub(r"[^a-zA-Z0-9]", " ", text.lower())
words = text.split()
print(words)
```

Output:

```
['natural', 'language', 'processing', 'is', 'an', 'exciting', 'area', 'huge',
'budget', 'have', 'been', 'allocated', 'for', 'this']
```

Stop Words removal: When we use the features from a text to model, we will encounter a lot of noise. These are the stop words like the, he, her, etc… which don't help us and, just be removed before processing for cleaner processing inside the model. With NLTK we can see all the stop words available in the English language.

Stemming: In our text we may find many words like playing, played, playfully, etc… which have a root word, play all of these convey the same meaning. So we can just extract the root word and remove the rest. Here the root word formed is called 'stem' and it is not necessarily that stem needs to exist and have a meaning. Just by committing the suffix and prefix, we generate the stems.

NLTK provides us with PorterStemmer LancasterStemmer and SnowballStemmer packages.

Input:

```
from nltk.stem.porter import PorterStemmer
# Reduce words to their stems
stemmed = [PorterStemmer().stem(w) for w in words]
print(stemmed)
```

Output:

```
['natur', 'languag', 'process', 'excit', 'area', 'huge', 'budget', 'alloc']
```

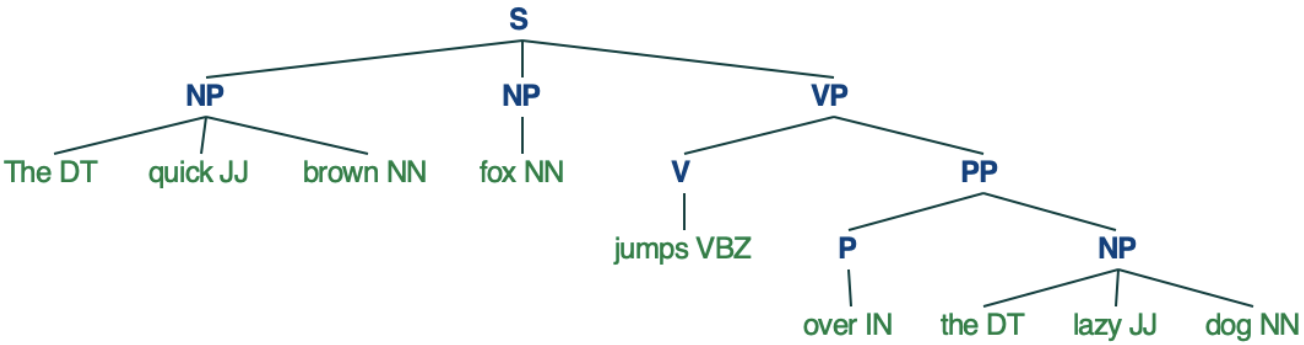
Lemmatization: We want to extract the base form of the word here. The word extracted here is called Lemma and it is available in the dictionary. We have the WordNet corpus and the lemma generated will be available in this corpus. NLTK provides us with the WordNet Lemmatizer that makes use of the WordNet Database to lookup lemmas of words.

```
from nltk.stem.wordnet import WordNetLemmatizer

# Reduce words to their root form

lemmed = [WordNetLemmatizer().lemmatize(w) for w in words]

print(lemmed)
```

Output:

```
['natural', 'language', 'processing', 'exciting', 'area', 'huge', 'budget',
'allocated']
```

Parse tree or Syntax Tree generation : We can define grammar and then use NLTK RegexpParser to extract all parts of speech from the sentence and draw functions to visualize it.



Conclusion: In the above experiment, we have learnt NLP made it possible for machines to become our friend.

Experiment 10

NLP Corpus

AIM: To study NLP corpus/ Treebank(e.g. Wordnet/ any other).

Description: A corpus is a large and structured set of machine-readable texts that have been produced in a natural communicative setting. Its plural is corpora. They can be derived in different ways like text that was originally electronic, transcripts of spoken language and optical character recognition, etc.

Elements of Corpus Design - Language is infinite but a corpus has to be finite in size. For the corpus to be finite in size, we need to sample and proportionally include a wide range of text types to ensure a good corpus design.

Corpus Representativeness -- Representativeness is a defining feature of corpus design. The following definitions from two great researchers − Leech and Biber, will help us understand corpus representativeness −

- **According to Leech (1991),** “A corpus is thought to be representative of the language variety it is supposed to represent if the findings based on its contents can be generalized to the said language variety”.
- **According to Biber (1993),** “Representativeness refers to the extent to which a sample includes the full range of variability in a population”.

In this way, we can conclude that representativeness of a corpus are determined by the following two factors −

- **Balance** − The range of genre include in a corpus
- **Sampling** − How the chunks for each genre are selected.

Corpus Balance - Another very important element of corpus design is corpus balance – the range of genre included in a corpus. We have already studied that representativeness of a general corpus depends upon how balanced the corpus is. We do not have any reliable scientific measure for balance but the best estimation and intuition works in this concern. In other words, we can say that the accepted balance is determined by its intended uses only.

Sampling - Another important element of corpus design is sampling. Corpus representativeness and balance is very closely associated with sampling. That is why we can say that sampling is inescapable in corpus building.

- According to Biber(1993), “Some of the first considerations in constructing a corpus concern the overall design: for example, the kinds of texts included, the number of texts, the selection of particular texts, the selection of text samples from within texts, and the length of text samples. Each of these involves a sampling decision, either conscious or not.”

Year	Name of the Corpus	Size (in words)
1960s - 70s	Brown and LOB	1 Million words
1980s	The Birmingham corpora	20 Million words
1990s	The British National corpus	100 Million words
Early 21st century	The Bank of English corpus	650 Million words

While obtaining a representative sample, we need to consider the following −

- Sampling unit − It refers to the unit which requires a sample. For example, for written text, a sampling unit may be a newspaper, journal or a book.
- Sampling frame − The list of al sampling units is called a sampling frame.
- Population − It may be referred as the assembly of all sampling units. It is defined in terms of language production, language reception or language as a product.

Experiment 11

TF/IDF

AIM: Program to calculate TF/IDF , feature extraction and finding unique words

- Feature Extraction

Input:

```
docA = "Jay Shri Ram"
docB = "Jay Shri Krishna"
docC = "Ram Krishna Paramhansa"
bowA = docA.split()
bowB = docB.split()
bowC = docC.split()
print(bowA)
print(bowB)
print(bowC)
```

Output:

```
['Jay', 'Shri', 'Ram']
['Jay', 'Shri', 'Krishna']
['Ram', 'Krishna', 'Paramhansa']
```

- Finding Unique Words

Input:

```
vocab = set(bowA).union(bowB).union(bowC)
vocab
```

Output:

```
{'Jay', 'Krishna', 'Paramhansa', 'Ram', 'Shri'}
```

Input:

```
wordDictA = dict.fromkeys(vocab, 0)
wordDictB = dict.fromkeys(vocab, 0)
wordDictC = dict.fromkeys(vocab, 0)

for word in bowA:
    wordDictA[word] += 1
for word in bowB:
    wordDictB[word] += 1
for word in bowC:
    wordDictC[word] += 1
print(wordDictA, wordDictB, wordDictC)
```

Output:

```
{'Paramhansa': 0, 'Krishna': 0, 'Shri': 1, 'Jay': 1, 'Ram': 1} {'Paramhansa':
0, 'Krishna': 1, 'Shri': 1, 'Jay': 1, 'Ram': 0} {'Paramhansa': 1, 'Krishna':
1, 'Shri': 0, 'Jay': 0, 'Ram': 1}
```

Input:

```
import pandas as pd
pd.DataFrame([wordDictA,wordDictB,wordDictC])
```

Output:

	Paramhansa	Krishna	Shri	Jay	Ram
0	0	0	1	1	1
1	0	1	1	1	0
2	1	1	0	0	1

- TF-IDF

Input:

```
def calcTF(wordDict, bow):
    tfDict = {}
    bowCount = len(bow)
    for word, count in wordDict.items():
        tfDict[word] = count/float(bowCount)
```

```
    return tfDict

tfBowA = calcTF(wordDictA, bowA)
tfBowB = calcTF(wordDictB, bowB)
tfBowC = calcTF(wordDictC, bowC)

print(tfBowA)
print(tfBowB)
print(tfBowC)
```

Output:

```
{'Paramhansa': 0.0, 'Krishna': 0.0, 'Shri': 0.3333333333333333, 'Jay':
0.3333333333333333, 'Ram': 0.3333333333333333}
{'Paramhansa': 0.0, 'Krishna': 0.3333333333333333, 'Shri':
0.3333333333333333, 'Jay': 0.3333333333333333, 'Ram': 0.0}
{'Paramhansa': 0.3333333333333333, 'Krishna': 0.3333333333333333, 'Shri':
0.0, 'Jay': 0.0, 'Ram': 0.3333333333333333}
```

- IDF

Input:

```
# IDF

import math
def calcIDF(docList):
    idfDict = {}
    N = len(docList)

    idfDict = dict.fromkeys(docList[0].keys(), 0)
    for doc in docList:
        for word, val in doc.items():
            if val > 0:
                idfDict[word] += 1

    for word, val in idfDict.items():
        idfDict[word] = math.log10((N + 1) / float(val) + 1.0) + 1.0

    return idfDict
idfs = calcIDF([wordDictA,wordDictB,wordDictC])
print(idfs)
```

Output:

```
{'Paramhansa': 1.6989700043360187, 'Krishna': 1.4771212547196624, 'Shri':
1.4771212547196624, 'Jay': 1.4771212547196624, 'Ram': 1.4771212547196624}
```

- TF/IDF

Input:

```
# TFIDF

def calcTFIDF(tfBow, idfs):
    tfidf = {}
    for word, val in tfBow.items():
        tfidf[word] = val*idfs[word]
    return tfidf
idfFirst = calcTFIDF(tfBowA, idfs)
idfSecond = calcTFIDF(tfBowB, idfs)
idfThird = calcTFIDF(tfBowC, idfs)
idf= pd.DataFrame([idfFirst, idfSecond,idfThird])
idf
```

Output:

	Paramhansa	Krishna	Shri	Jay	Ram
0	0.000000	0.000000	0.492374	0.492374	0.492374
1	0.000000	0.492374	0.492374	0.492374	0.000000
2	0.566323	0.492374	0.000000	0.000000	0.492374

Experiment 12

Word2Vec

AIM: Convert Word to Vector

Code:

- Data Collection

Input:

```
corpus = ['I like apple juice',
          'I like orange juice',
          'king is a strong man',
          'queen is a wise woman',
          'boy is a young man',
          'girl is a young woman',
          'prince is a young king',
          'princess is a young queen',
          'man is strong',
          'woman is pretty',
          'prince is a boy will be king',
          'princess is a girl will be queen',
         'Apple is good place for work']
print(corpus)
```

Output:

```
['I like apple juice', 'I like orange juice', 'king is a strong man', 'queen is a wise woman', 'boy is a young man', 'girl is a young woman', 'prince is a young king', 'princess is a young queen', 'man is strong', 'woman is pretty', 'prince is a boy will be king', 'princess is a girl will be queen', 'Apple is good place for work']
```

- Remove Stop Words

Input:

```
# Preprocessing steps
def remove_stop_words(corpus):
```

```
    stop_words = ['is', 'a', 'will', 'be']
    results = []
    for text in corpus:
        tmp = text.split(' ')
        for stop_word in stop_words:
            if stop_word in tmp:
                tmp.remove(stop_word)
        results.append(" ".join(tmp))

    return results

# After removing all stop-words
corpus = remove_stop_words(corpus)
print(corpus)
```

Output:

```
['I like apple juice', 'I like orange juice', 'king strong man', 'queen wise
woman', 'boy young man', 'girl young woman', 'prince young king', 'princess
young queen', 'man strong', 'woman pretty', 'prince boy king', 'princess girl
queen', 'Apple good place for work']
```

Input:

```
words = []
for text in corpus:
    for word in text.split(' '):
        words.append(word)
type(words)
print(words)
```

Output:

```
list
['I', 'like', 'apple', 'juice', 'I', 'like', 'orange', 'juice', 'king',
'strong', 'man', 'queen', 'wise', 'woman', 'boy', 'young', 'man', 'girl',
'young', 'woman', 'prince', 'young', 'king', 'princess', 'young', 'queen',
'man', 'strong', 'woman', 'pretty', 'prince', 'boy', 'king', 'princess',
'girl', 'queen', 'Apple', 'good', 'place', 'for', 'work']
```

- Data Generation

Input:

```
word2int = {}

#Here we assisgned number to each word store it into Dictionary
for i,word in enumerate(words):
    word2int[word] = i

# Here we split corpus into sentences
sentences = []
for sentence in corpus:
    sentences.append(sentence.split())

WINDOW_SIZE = 2 # Dimension is 2 means "we consider 2 words from left and
right to the centre word

data = []
for sentence in sentences:
    for idx, word in enumerate(sentence):
        for neighbor in sentence[max(idx - WINDOW_SIZE, 0) : min(idx +
WINDOW_SIZE, len(sentence)) + 1] :
            if neighbor != word:
                data.append([word, neighbor])

import pandas as pd
df = pd.DataFrame(data, columns = ['input', 'label'])
print(df.head())
```

Output:

	input	label
0	I	like
1	I	apple
2	like	I
3	like	apple
4	like	juice
...	...	...
81	for	good
82	for	place
83	for	work

	input	label
84	work	place
85	work	for

86 rows x 2 columns

Input:

```
df.shape
word2int
len(words)
```

Output:

```
(86, 2)
{'Apple': 36, 'I': 4, 'apple': 2, 'boy': 31, 'for': 39, 'girl': 34, 'good':
37, 'juice': 7, 'king': 32, 'like': 5, 'man': 26, 'orange': 6, 'place': 38,
'pretty': 29, 'prince': 30, 'princess': 33, 'queen': 35, 'strong': 27,
'wise': 12, 'woman': 28, 'work': 40, 'young': 24}
41
```

- Define Tensorflow Graph

Input:

```
import tensorflow as tf
import numpy as np

ONE_HOT_DIM = len(words)

# function to convert numbers to one hot vectors
def to_one_hot_encoding(data_point_index):
    one_hot_encoding = np.zeros(ONE_HOT_DIM)
    one_hot_encoding[data_point_index] = 1
    return one_hot_encoding

X = [] # input word
Y = [] # target word

for x, y in zip(df['input'], df['label']):
    X.append(to_one_hot_encoding(word2int[ x ]))
```

```
    Y.append(to_one_hot_encoding(word2int[ y ]))

# convert them to numpy arrays
X_train = np.asarray(X)
Y_train = np.asarray(Y)

X_train[0]
ONE_HOT_DIM
```

Output:

```
array([0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
0., 0., 0., 0., 0.])
41
```

Input:

```
# # import Tensorflow 2
import tensorflow as tf

# placeholders are not executable immediately so we need to disable eager
exicution in TF 2 not in 1
tf.compat.v1.disable_eager_execution()

# # Create Placeholder
# making placeholders for X_train and Y_train
x = tf.compat.v1.placeholder(tf.float32, shape=(None, ONE_HOT_DIM))
#x=v1.placeholder(tf.float32,shape=(None,ONE_HOT_DIM))
y_label = tf.compat.v1.placeholder(tf.float32, shape=(None, ONE_HOT_DIM))


# word embedding will be 2 dimension for 2d visualization
EMBEDDING_DIM = 2

# hidden layer: which represents word vector eventually
W1 = tf.Variable(tf.random.normal([ONE_HOT_DIM, EMBEDDING_DIM]))
b1 = tf.Variable(tf.random.normal([1])) #bias
hidden_layer = tf.add(tf.matmul(x,W1), b1)

# output layer
W2 = tf.Variable(tf.random.normal([EMBEDDING_DIM, ONE_HOT_DIM]))
b2 = tf.Variable(tf.random.normal([1]))
prediction = tf.nn.softmax(tf.add( tf.matmul(hidden_layer, W2), b2))
```

```
# loss function: cross entropy
loss = tf.reduce_mean(-tf.reduce_sum(y_label * tf.math.log(prediction), axis=
[1]))

# training operation
train_op = tf.compat.v1.train.GradientDescentOptimizer(0.05).minimize(loss)
```

- Train

Input:

```
sess = tf.compat.v1.Session()
init = tf.compat.v1.global_variables_initializer()
sess.run(init)

iteration = 20000
for i in range(iteration):
    # input is X_train which is one hot encoded word
    # label is Y_train which is one hot encoded neighbor word
    sess.run(train_op, feed_dict={x: X_train, y_label: Y_train})
    if i % 3000 == 0:
        print('iteration '+str(i)+' loss is : ', sess.run(loss, feed_dict={x:
X_train, y_label: Y_train}))

# Now the hidden layer (W1 + b1) is actually the word look up table
vectors = sess.run(W1 + b1)
print(vectors)
```

Output:

```
iteration 0 loss is : 4.4621043 iteration 3000 loss is : 1.9002442 iteration
6000 loss is : 1.6928054 iteration 9000 loss is : 1.6478184 iteration 12000
loss is : 1.6222945 iteration 15000 loss is : 1.6065267 iteration 18000 loss
is : 1.5952846

[[-0.25849512 1.0124775 ] [ 1.2583842 0.24572852] [ 4.094598 -0.679459 ]
[-0.26620525 -0.12204699] [ 2.870049 1.89985 ] [ 2.2995784 -0.49068558] [
4.146911 -0.68337786] [ 2.8695571 1.8998017 ] [-0.21131328 -0.2777498 ]
[-0.8767552 -0.87916774] [-0.0468364 1.4572078 ] [-1.3502573 0.44072923]
[-0.13251948 3.0822113 ] [-1.8298188 -0.45694157] [-0.3163423 -1.4209694 ] [
0.98468065 -0.05145909] [ 0.30405837 0.6912785 ] [ 1.1939338 0.47238547]
[-0.70663935 1.1642432 ] [-0.60522276 0.27503708] [ 1.1305941 1.7720782 ] [
0.1997036 1.1094276 ] [-1.4043299 0.27796325] [ 1.8406576 -1.3036556 ]
```

```
[-0.81767213 0.5417353 ] [-0.32076916 -1.9922065 ] [-2.6566114 0.5042405 ]
[-3.874454 -3.2495613 ] [-1.0156348 1.321537 ] [ 1.0666234 3.5757337 ]
[-4.94816 0.79142404] [-2.1972554 0.11522846] [-1.628582 0.12392661]
[-3.078818 3.7606344 ] [-2.344927 2.9857001 ] [-1.6083901 1.9211624 ]
[-0.35790554 -3.898341 ] [ 1.649157 -3.0048156 ] [ 0.04651211 -1.6022413 ]
[-0.8088883 -3.5907106 ] [ 1.8440939 -3.7960815 ]]
```

- Word2Vec

Input:

```
type(vectors),type(words)
data1={}
w2v_df = pd.DataFrame(vectors, columns = ['x1', 'x2'])
w2v_df['word'] = words
w2v_df = w2v_df[['word', 'x1', 'x2']]
w2v_df.head()
```

Output:

```
(numpy.ndarray, list)
```

	word	x1	x2
0	I	-0.258495	1.012478
1	like	1.258384	0.245729
2	apple	4.094598	-0.679459
3	juice	-0.266205	-0.122047
4	I	2.870049	1.899850

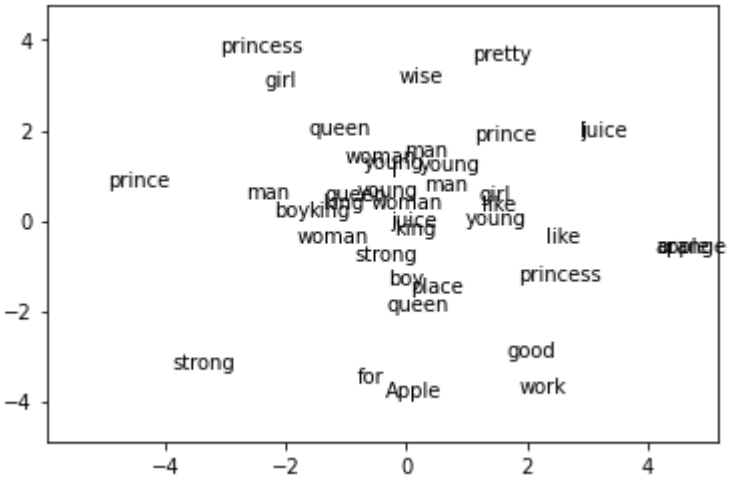
- Word2Vec in 2D Chart

Input:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

for word, x1, x2 in zip(w2v_df['word'], w2v_df['x1'], w2v_df['x2']):
    ax.annotate(word, (x1,x2 ))
```

```
PADDING = 1.0
x_axis_min = np.amin(vectors, axis=0)[0] - PADDING
y_axis_min = np.amin(vectors, axis=0)[1] - PADDING
x_axis_max = np.amax(vectors, axis=0)[0] + PADDING
y_axis_max = np.amax(vectors, axis=0)[1] + PADDING

plt.xlim(x_axis_min,x_axis_max)
plt.ylim(y_axis_min,y_axis_max)
plt.rcParams["figure.figsize"] = (15,15)

plt.show()
```

Output:



Experiment 13

Out-of-Vocablary

AIM: Padding and Out-of-Vocablary

Code:

- Convert Words into tokens

Input:

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.preprocessing.text import Tokenizer
sentences = ['I love my country', 'I love my religion' ]

token = Tokenizer(num_words = 100)
token.fit_on_texts(sentences)
word_index=token.word_index
print(word_index)

samvidhan = ["Right to equality",
             "Right to freedom" ,
             "Right against exploitation",
             "Right to freedom of religion",
             "Cultural and educational rights",
             "Right to constitutional remedies"]

token = Tokenizer(num_words = 100)
token.fit_on_texts(samvidhan)
word_index=token.word_index
print(word_index)

sequence = token.texts_to_sequences(samvidhan)
print(sequence)

asharfi = ["Teri Jhalak Asharfi Srivalli",
            "Naina Madak Barfi",
            "Teri Jhalak Asarfi Srivalli",
            "Baatein Kare Jo Harfi"]
token = Tokenizer(num_words = 100)
token.fit_on_texts(asharfi)
word_index=token.word_index
```

```
print(word_index)

song = ["Mujhko Hai Bass Tera Banna",
        "Ek Jhalak Teri Ankon Me",
        "Khwaab Saja Jaaye",
        "Teri Jhalak Asharfi Srivalli",
        "Naina Maadak Barfi",
        "Teri Jhalak Asharfi Srivalli",
        "Batein Kare Do Harfi"]
token = Tokenizer(num_words = 100)
token.fit_on_texts(song)
word_index=token.word_index
print(word_index)

sequence = token.texts_to_sequences(song)
print(sequence)

asharfi_result = token.texts_to_sequences(asharfi)
print(asharfi_result)
```

Output:

```
{'i': 1, 'love': 2, 'my': 3, 'country': 4, 'religion': 5}

{'right': 1, 'to': 2, 'freedom': 3, 'equality': 4, 'against': 5,
'exploitation': 6, 'of': 7, 'religion': 8, 'cultural': 9, 'and': 10,
'educational': 11, 'rights': 12, 'constitutional': 13, 'remedies': 14}

[[1, 2, 4], [1, 2, 3], [1, 5, 6], [1, 2, 3, 7, 8], [9, 10, 11, 12], [1, 2,
13, 14]]

{'teri': 1, 'jhalak': 2, 'srivalli': 3, 'asharfi': 4, 'naina': 5, 'madak': 6,
'barfi': 7, 'asarfi': 8, 'baatein': 9, 'kare': 10, 'jo': 11, 'harfi': 12}
{'jhalak': 1, 'teri': 2, 'asharfi': 3, 'srivalli': 4, 'mujhko': 5, 'hai': 6,
'bass': 7, 'tera': 8, 'banna': 9, 'ek': 10, 'ankon': 11, 'me': 12, 'khwaab':
13, 'saja': 14, 'jaaye': 15, 'naina': 16, 'maadak': 17, 'barfi': 18,
'batein': 19, 'kare': 20, 'do': 21, 'harfi': 22} [[5, 6, 7, 8, 9], [10, 1, 2,
11, 12], [13, 14, 15], [2, 1, 3, 4], [16, 17, 18], [2, 1, 3, 4], [19, 20, 21,
22]] [[2, 1, 3, 4], [16, 18], [2, 1, 4], [20, 22]]
```

- Padding Out-of-Vocabulary

Input:

```
from tensorflow.keras.preprocessing.sequence import pad_sequences
token = Tokenizer(num_words = 100, oov_token = "<oov>")
token.fit_on_texts(asharfi)
word_index = token.word_index
sequeuces = token.texts_to_sequences(asharfi)
padded = pad_sequences(sequeuces)
print(word_index)
print(sequeuces)
print(padded)

test_data = [ "Teri Jhalak Asharfi Srivalli",
              "Naina Madak Barfi",
              "Teri Jalak Asharfi Srivalli",
              "Baatein Kare Jo Harfi"]

test_seq = token.texts_to_sequences(test_data)
print(test_seq)

padded = pad_sequences(sequence, padding= 'post', maxlen = 10)

print(word_index)
print(sentences)
print(padded)

test_seq = token.texts_to_sequences(test_data)
test_padded = pad_sequences(test_seq, padding= 'post', maxlen= 10)
print(test_padded)
```

Output:

```
{'<oov>': 1, 'teri': 2, 'jhalak': 3, 'srivalli': 4, 'asharfi': 5, 'naina': 6,
'madak': 7, 'barfi': 8, 'asarfi': 9, 'baatein': 10, 'kare': 11, 'jo': 12,
'harfi': 13} [[2, 3, 5, 4], [6, 7, 8], [2, 3, 9, 4], [10, 11, 12, 13]] [[ 2 3
5 4] [ 0 6 7 8] [ 2 3 9 4] [10 11 12 13]]

[[2, 3, 5, 4], [6, 7, 8], [2, 1, 5, 4], [10, 11, 12, 13]]

{'<oov>': 1, 'teri': 2, 'jhalak': 3, 'srivalli': 4, 'asharfi': 5, 'naina': 6,
'madak': 7, 'barfi': 8, 'asarfi': 9, 'baatein': 10, 'kare': 11, 'jo': 12,
'harfi': 13} ['I love my country', 'I love my religion'] [[ 5 6 7 8 9 0 0 0 0
0] [10 1 2 11 12 0 0 0 0 0] [13 14 15 0 0 0 0 0 0 0] [ 2 1 3 4 0 0 0 0 0 0]
[16 17 18 0 0 0 0 0 0 0] [ 2 1 3 4 0 0 0 0 0 0] [19 20 21 22 0 0 0 0 0 0]]

[[ 2 3 5 4 0 0 0 0 0 0] [ 6 7 8 0 0 0 0 0 0 0] [ 2 1 5 4 0 0 0 0 0 0] [10 11
12 13 0 0 0 0 0 0]]
```