B-Tech Project Report

To develop accurate POS Tagger for Hindi language

under guidance of

Prof. Gaurav Harit

Computer Science and Engineering
Indian Institute of Technology, Jodhpur

Shashwat Kathuria
(B17CS050)
II Year Undergraduate
Computer Science and Engineering
Indian Institute of Technology, Jodhpur

Shreyas Mahajan
(B17CS051)
II Year Undergraduate
Computer Science and Engineering
Indian Institute of Technology, Jodhpur

Department of Computer Science and Engineering
Indian Institute of Technology, Jodhpur
April 2019

Certificate

It is certified that the work contained in the project report titled **"Efficient POS Tagger for Hindi Language"** by **Shashwat Kathuria** and **Shreyas Mahajan** has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Prof. Gaurav Harit
Dept. of Computer Science and Engineering
Indian Institute of Technology, Jodhpur
April 2019

Declaration

I declare that this project submission represents my ideas in my own words. Wherever others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented, fabricated or falsified any idea, data, fact, or source in my submission.

Shashwat Kathuria
II Year Undergraduate
Computer Science and Engineering
Indian Institute of Technology, Jodhpur

Shreyas Mahajan
II Year Undergraduate
Computer Science and Engineering
Indian Institute of Technology, Jodhpur

Acknowledgements

We would like to express my sincere gratitude to our project supervisor **Prof. Gaurav Harit** for motivating us in taking such a crucial project, for helping by giving valuable suggestions, comments and proper guidance throughout the course of the project.

We thank distributors of various modules of scikit learn, nltk and other packages used in project. Also thanks to collectors of Hindi language tagged POS corpus and to Penn Treebank Tagset for providing universal list and abbreviations for POS tags used.

Abstract

In the world of Natural Language Processing (NLP), the most basic models are based on Bag of Words (BOW), which fail to capture the syntactic relations between words.

Assigning Parts of Speech (POS) is a basic function when it comes to making various text analysing modules and functionalities. Many software artifacts are written in natural language or contain substantial amount of natural language contents. Thus these could be analyzed using text analysis techniques from the NLP. Part of Speech Tags are useful for building parse trees, which are used in building NERs (most named entities are Nouns) and extracting relations between words. POS Tagging is also essential for building lemmatizers which are used to reduce a word to its root form. To understand the meaning of any sentence or to extract relationships and build a knowledge graph, POS Tagging is a very important step. Various approaches are implemented and tested for english corpus previously.

But, for hindi language, very little amount of work is done in this field. Hindi is morphologically very strong language. This task is not straightforward, as a particular word may have a different part of speech based on the context in which the word is used. For example, a model will not be able to capture the difference between “इस होटल में अच्छा खाना मिलता है।”, where “खाना” is a noun, and “पैसा खाना बुरी बात है |”, where “खाना” is a verb.

We have gone through various approaches towards POS tagging and studied them sincerely. We also went through some techniques used in English POS Tagging and thought on how we could integrate it for Hindi. Through this report, we present one approach to solve the problem of POS tagging and also provide analysis and comparison stats with other possible and/or existing techniques for Hindi corpus.

Table of Contents

1. Introduction

1.1 Parts of Speech

Every language defines its grammar to categorize words according to its sense and contextual use. Words are spoken or written together in a specific manner to create meaningful sentences, irrespective of language, and thus the interpretation of the message can be changed in how the words are arranged or used. In order to get the proper sequence of words to deliver the correct message, proper understanding of different categories or parts of speech is a must.

As in English, Hindi also defines following Parts of Speech:

1) Noun (Sangya)
2) Pronoun (Sarvanam)
3) Adjectives (Vesheshan)
4) Verb (Kriya)
5) Adverb (Kriya Vesheshan)
6) Preposition (Sambandhbodhak)
7) Conjuction (Sammuchyabodhak)
8) Interjection (Vismayadibodhak)

The part of speech indicates how the word functions in meaning as well as grammatically within the sentence. An individual word can function as more than one part of speech when used in different circumstances. This is a helpful way to understand the underlying grammar and logic of any language under study.

1.2 POS Tagging

POS Tagging, also called word-category disambiguation, is the process of tagging words in a sentence to its appropriate Part of Speech according to the function of the word to make the sentence grammatically correct. This is done considering both its definition, as well as context of the word, which is analysed by its relationship with adjacent or related words in a phrase, sentence, paragraph or even a document.

This is achieved by performing mainly two things, starting with transforming textual information in numerical format, and then decoding this information to select appropriate tag.

1.3 Importance of POS Tagging

Uses of part-of-speech tagging algorithms vary depending on the field(s) where they are applied. It is one of the most foremost functionality used oftenly in Natural Language Processing Applications. However, main benefits of POS Tagging are information retrieval, indexing and classification.

One can deduct main objects, phrases, actions in articles, tweets or any piece of information. It finds its use in many Text-to-Speech (TTS) applications, information extraction, linguistic research for corpora, and also can be used as an intermediate step for higher level NLP tasks such as parsing, semantics analysis, translation, and many more. It is also used to build the knowledge base of a Natural Language Analyzer.

1.4 Hindi POS Tagging

Written in Devanagari script, Hindi is one of the official and standardized language spoken in India by over 260 million native speakers. As a linguistic variety, Hindi is the fourth most-spoken first language in the world.

Despite such popularity, very few work is done till now in developing efficient POS Taggers for the language. Many people use Hindi as a preferred languages on internet in forms of articles, tweets, blogs etc and has a huge vocabulary too. Thus efficient text analysers for Hindi language are a must.

1.5 Complexities

Hindi is morphologically strong language. Due to its huge vocabulary and complex grammar involved, setting up a set of rules to assign POS tags is nearly impossible. Various words get difficult to categorize and require cleaning of the data first. Defining such a set of rules is also a tedious task. Moreover as vocabulary is large, it may happen that a word is not present in data or it is present in data in some other form (mostly by changed prefix/suffix). Hence there should be a technique to analyse such conditions.

We concentrate on following important examples to get an idea of the complexities involved.

Example 1:
Case where same word is used as different context, thus getting different POS tag assigned.

(1) “इस होटल में अच्छा ***खाना*** मिलता है।”

Translation: Good food is available in this hotel.

(2) “पैसा ***खाना*** बुरी बात है |”

Translation: Eating money (corruption) is a bad practice.

In sentence 1, word “खाना” means food, which makes it a noun. Whereas in sentence 2, it means to eat, which makes it a verb. This is an important but difficult task to differentiate its contextual meaning and use according to the sentence.

Example 2:
Case when a specific word is used by changing/updating its prefix or suffix

(1) काबू - बेकाबू

(2) लिखेगा - लिखेगी - लिखेंगे

(3) विज्ञान - मनोविज्ञान

In category 1, prefix “बे” changes the meaning from “control” to “without control” which makes it opposite of each other whereas in category 3, prefix “मनो” changes the meaning from “science” to “psychology (mind science)” which adds more detail to same word.
In category 2, suffixes “गा”, “गी”, “गे” are different according to tenses, but meaning of the words is the same.

2. Background

2.1 Literature review

Various different approaches have been tried, implemented and tested across many languages. These are clubbed in categories based on similarity and then followed by various decoding algorithms to get the results we desire. Some techniques are evolved from previous ones whereas some are combined with other techniques to obtain good results. Following diagram signifies most of these which are used efficiently.

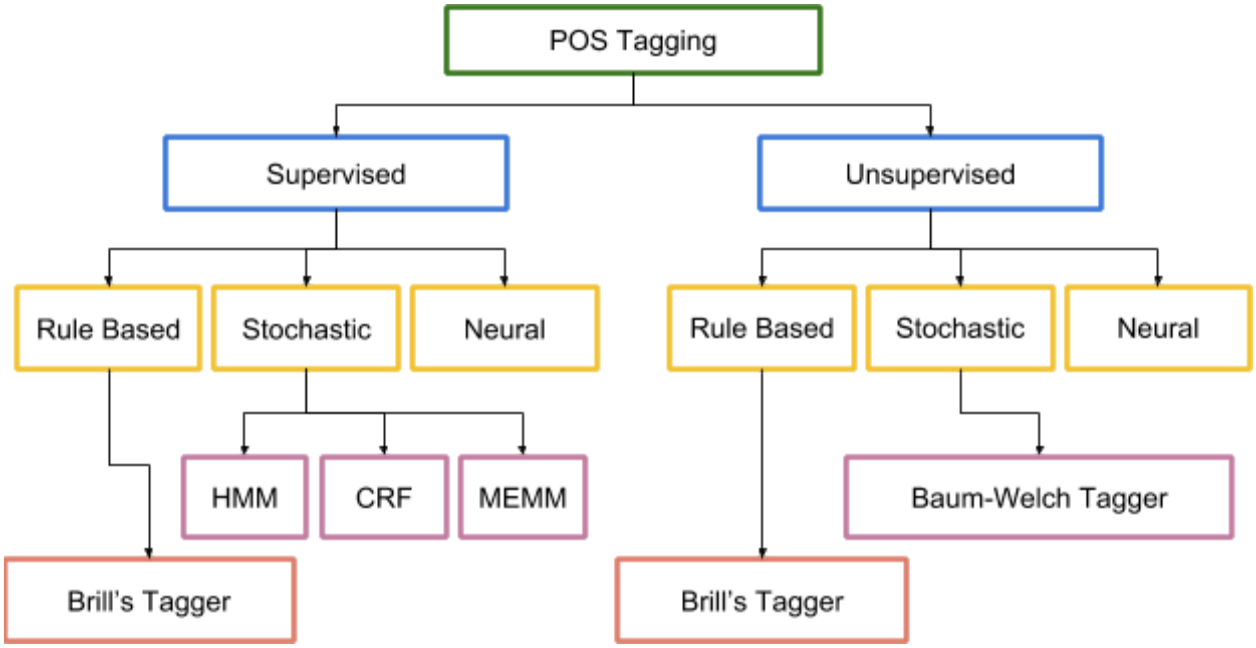


Diagram: Methodologies available and/or adopted for POS Tagging

2.1.1 Supervised vs Unsupervised

This is the first and the foremost distinction made amongst the POS taggers which is made on the basis of degree of automation of training and tagging processes. Supervised taggers typically rely on pre-tagged corpora and serve as the basis for creating any tools to be used. Example: The word/tag frequencies, the tag sequence probabilities and/or the Rule set. Unsupervised models, on the other hand, are those which do not require a pre-tagged corpus but instead use sophisticated computational methods to automatically induce word groupings.

This helps in calculating the probabilistic information needed by stochastic taggers and to induce the context rules needed by rule-based systems.

Key differences between two approaches	
Supervised	Unsupervised
1. Selection of tagged corpus. 2. Creation of dictionaries using tagged corpus. 3. Uses off line analysis. 4. Number of classes are known.	1. Induction of tagset using untagged training data. 2. Induction of dictionary using training data. 3. Can use real time data. 4. Number of classes are not known.

2.1.2 Rule Based Taggers

Typically rule based approaches use particular, common information to assign tags to unknown/ambiguous words. Some of the type of rules are discussed below.

Type 1:
Contextual Information
This type of rule deals with tags normally preceding or succeeding specific tag or group of tags. Thus it defines the tag considering tags surrounding the word.
Example: Word ‘X’ preceded by a determiner and followed by a noun, tag it as an adjective.

Type 2:
Morphological Information
This type of rule deals with tags associated with words having common suffixes or prefixes or both. Thus it is defined not by its surrounding tags but by the word itself.
Example: Word ‘X’ ending in ‘-ing’, tag it as a verb.

Type 3:
Capitalization and Punctuation
This type of rule deals with tags for words starting with capital letters and ending with certain punctuation marks. This information is of greater or lesser value depending on the language selected.
Example: German uses capitalization to predict nouns, whereas Hindi cannot use.

2.1.3 Stochastic Taggers

Any model which somehow incorporates statistics in it can be referred as stochastic. It can thus provide various approaches by selecting different types of probabilistic details fetched from the corpus in right ratio to give desired results. Following are some of the probabilities considered.

Type 1:
Word frequency measurements
This is a simple way that disambiguate words based on probability of its occurrence with a particular tag. The tag encountered most frequently in the training set is the one assigned to an ambiguous instance of that word.

Type 2:
Tag sequence probabilities
This refers to the fact that the best tag for a word is determined by probability it occurs along with previous n-1 tags. This is also termed as n-gram approach. However, it requires efficient decoding algorithms for determining the tag.

2.2 Literature Survey

2.2.1 State-of-Art Approaches

TnT Tagger:
Trigrams'n'Tags (TnT) is a very efficient statistical POS tagger, trainable on different languages. The component for parameter generation trains on tagged corpora. Moreover, it incorporates several methods of smoothing and handling unknown/ambiguous words. Additionally, TnT Tagger is optimized for speed too. TnT comes for two major language models, English and German. The German model is trained on the Saarbrücker German newspaper corpus using the Stuttgart-Tübingen-Tagset. The English model is trained on the Susanne Corpus.
TnT is an implementation of Viterbi algorithm for second order Markov models and the main paradigm used for smoothing here is linear interpolation (the respective weights are determined by deleted interpolation). Unknown words are worked out by a suffix trie and successive abstraction.

Highlights
Language: English
Corpus: Susanne corpus/Penn Treebank (Domain: Newspaper)
Size: 1,200,000 (Penn) 150,000 (Susanne)
Accuracy: 96.7% (Penn) 94.5% (Susanne)
Reference: *http://www.coli.uni-saarland.de/~thorsten/tnt/*

GENIA Tagger:
The GENIA tagger analyzes English sentences and outputs base forms, POS tags, chunk tags, and named entity tags. The tagger is specifically tuned for biomedical text such as 'Medicine' abstracts. This tagger is therefore a useful preprocessing tool to extract information from biomedical documents.

Highlights
Language: English
Corpus: Wall Street Journal & GENIA corpus (Domain: Biomedicine)
Accuracy: 97.05% (WSJ, trained on WSJ) 85.19% (GENIA, trained on WSJ)
Reference: *http://www.nactem.ac.uk/tsujii/GENIA/tagger/*

Stanford Tagger 2.0:
Stanford POS Tagger is a software distributed by 'stanford.edu'. It is a Java implementation of log-linear POS Taggers. It uses different computational approaches which require fine grained POS tags. It also describes feature-rich POS tagging using cyclic dependency networks. It provides tagger for Arabic, German, French, Chinese also, using Penn Treebank tagset.

Highlights
Language: English
Tokens: 912,344 (Training) 131,768 (Develop) 129,654 (Testing)
Accuracy: 97.29% (Tokens) 89.7% (Unknown words)
Reference: *https://nlp.stanford.edu/software/tagger.shtml*

2.2.2 Extension for Hindi POS Tagging

A POS tagging approach based on Maximum Entropy Markov Model using supervised training. This model trains using a pre-tagged corpora and uses a feature set to predict the tag for a word. It has achieved accuracy of 89.34% on the development data of the NLPAI Competition 2006.

A stochastic model based on Conditional Random Fields (CRF) is developed which demonstrates a performance of 82.67% accuracy.

A HMM based tagger, reporting a performance of 76.49% accuracy on training and test data having about 25000 and 6000 words, respectively. This tagger uses HMM in combination with probability models of certain contextual features for POS tagging.

A simple HMM based POS tagger, which employs a naive (longest suffix matching) stemmer as a pre-processor to achieve reasonably good accuracy of 93.12%. This method does not require any linguistic resource apart from a list of possible suffixes for the language.

Part Of Speech (POS) tagging and Chunking using Conditional Random Fields (CRFs) and Transformation Based Learning (TBL) gave 78.66% accuracy for Hindi. It was extended to Telugu and Bengali also. Improved training methods based on the morphological information, contextual and the lexical rules (developed using TBL) were critical in achieving good results.

2.3 Dataset

The models are trained and tested on tagged dataset according to the universal tagging rules discussed above in the report. Following is the distribution of data of testing and training :-

Viterbi's Algorithm:

Corpus size:

431092 words, 18489 sentences

Training:

396712 words, 16993 sentences

Testing:

34380 words, 1496 sentences

Decision Tree:

Corpus size:

163433 words, 6993 sentences

Training:

122567 words, 5244 sentences

Testing:

40866 words, 1749 sentences

2.4 Universal Tagging Methodology

The following table describes tags which are included in our Hindi dataset. It follows Universal Tagging Methodology as in Penn Treebank Tagset at the University of Pennsylvania for English.

Tag	Meaning	Tag	Meaning
JJ	Adjective	NEG	Negative Word
RP	Particles	NN	Common Noun
PRP	Pronoun	RB	Adverb
INTF	Intensifier	RDP	Reduplications
VM	Main verb	UNK	Unknown words
INJ	Interjection	QF	Quantifiers

Tag	Meaning	Tag	Meaning
DEM	Demonstrative	WQ	Question Words
ECH	Echo Words	CC	Conjuncts
NST	Spatial Nouns	NNP	Proper Noun
VAUX	Verb Auxilary	XC	Compounds
SYM	Symbol	QC	Cardinals
PSP	Postposition	QO	Ordinals

3. Methodology and Algorithms

3.1 Proceedings

From the literature survey and review, as discussed in ‘Section 2’, for efficient POS taggers available, we decide to go with stochastic model approach. Rule based models will require a lot of work for Hindi as set of rules are still to be constructed so as to operate on computational algorithms. This provides a clear advantage for Hindi as probabilities and frequencies can be calculated without much hesitation. Moreover, data can be improved over again after observing the frequencies, thus making approach more efficient.

3.2 Hidden Markov Model

The Hidden Markov Model (HMM) is a sequence model, a model whose job is to assign a label or class to each unit in a sequence, also termed as sequence classifier, thus mapping a sequence of observations to a sequence of labels. More specifically, HMM is a probabilistic sequence model. Given a sequence of units (words in sentence/paragraphs in our case), it first computes a probability distribution over possible sequences of labels and then chooses the best label sequence.

HMM is a combination of following two approaches:

- Word sequence measurements
- Tag sequence probabilities

3.2.1 Generative Models and Noisy Channel Models

Supervised problems in machine learning are defined as follows.
Assume training examples (x(1), y(1)). . .(x(m), y(m)), where x(i) is input paired with a label y(i) .
Let X be used to refer to the set of possible inputs, and Y to refer to the set of possible labels.
Task: To learn a function f : X → Y which maps any input x from X to a label f(x) in Y.

One way to define the function f(x) is through a conditional model, wherein we define a model to define the conditional probability p(y|x) for any pair x, y. The parameters of the model are estimated from the training examples.
The output from the model is f(x) = arg max p(y|x) over all y in Y.
Thus in simple words, we take the most likely label y as the output from the model.

Another approach is to define a generative model. This is often used in Natural Language Processing. Instead of directly estimating the conditional probability distribution p(y|x), we instead find the joint probability p(x, y) over (x, y) pairs. This model is called Generative Model. The parameters of the model p(x, y) are again estimated from the training examples (x(i), y(i)) for i belonging to [1, n].

Further decomposition gives p(x, y) = p(y)*p(x|y)
These two model components have the following interpretations:

- p(y) : prior probability distribution over labels y.
- p(x|y) : probability of generating the input x, given that the label is y.

Use of Bayes' rule:
Given a generative model, we can use Bayes rule to derive the conditional probability p(y|x) which is as follows:

$$p(y|x) = \frac{p(y)p(x|y)}{p(x)}$$

$$p(x) = \sum_{y \in \mathcal{Y}} p(x, y) = \sum_{y \in \mathcal{Y}} p(y)p(x|y)$$

Thus the joint model is quite versatile as we can also derive other two probabilities, p(x) and p(y|x).

$$\begin{aligned} f(x) &= \arg\max_y p(y|x) \\ &= \arg\max_y \frac{p(y)p(x|y)}{p(x)} \\ &= \arg\max_y p(y)p(x|y) \end{aligned}$$

Models which decompose a joint probability distribution p(x, y) into terms of p(y) and p(x|y) are often called noisy-channel models. The model p(x|y) can be interpreted as a "channel" which takes a label y as its input, and retrieves x as its output.

3.2.2 Markov Chain Model

The Markov property suggests that the distribution for a random variable in the future depends solely only on its distribution in the current state, and none of the previous states have any impact on the future states.

$$P(q_1, \ldots, q_n) = \prod_{i=1}^{n} P(q_i|q_{i-1})$$

A Markov chain is a model that tells us information regarding the probabilities of sequences of random variables, states, each of which can take on values from some set. These sets can be words or tags or symbols representing anything. In short, all the states before the current state have no impact on the future states.

3.2.3 Definition of HMM

A hidden Markov model (HMM) provides us information about both observed events (words that are give as input) and hidden events (POS tags from tagset) that we think of as causal factors in our probabilistic model.

We define a HMM by the following components:

- A finite set of N states
- A sequence of T observations
- Initial probability distribution
- Probabilistic automata:
  - Emission probabilities, which represent the probabilities of making certain observations given a particular state.
  - Transition probabilities, which represent the probability of transitioning to another state given a particular state.

Let S to be the set of all sequence/tag-sequence pairs <x(1) . . . x(n), y(1) . . . y(n+1)>
such that n ≥ 0, x(i) ∈ V for i in [1, n], y(i) ∈ K for i in [1, n], and y(n+1) = STOP.
We then define the probability for any <x(1) . . . x(n), y(1) . . . y(n+1)> ∈ S as

$$p(x_1 \dots x_n, y_1 \dots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i|y_{i-2}, y_{i-1}) \prod_{i=1}^{n} e(x_i|y_i)$$

where:

q(y(i) | y(i-2), y(i-1)) is transition probability for trigram HMM (n-gram HMM where n=3)
e(x(i) | y(i)) is emission probability
y(0) = y(−1) = * (Assumed for the start of chain)

3.2.4 Independence Assumptions

Contextual Independence:
The probability of a tag depends on the previous one (bigram model) or previous two (trigram model) or in general previous n tags (n-gram model) The beginning of a sentence can be accounted for by assuming an initial probability for each tag.

$$P(t_1 \cdots t_m) = \prod_{i=1}^{m} P(t_i \mid t_{i-(n-1)} \cdots t_{i-1})$$

Lexical Independence:
The conditional probability can be approximated by assuming that a word appears in a category independent of the words in the preceding or succeeding categories. It is known as Word Generation Probability.

$$P(w_1 \cdots w_m \mid t_1 \cdots t_m) = \prod_{i=1}^{m} P(w_i \mid t_i)$$

3.3 Decoding

3.3.1 Brute-force approach

To find the maximum probabilistic sequence, the naive, brute-force method is simply to enumerate all possible tag sequences y(1) . . . y(n+1), score them under the function p, and take the highest scoring sequence. However, for longer sentences, this method will be inefficient give its exponential growth with respect to n.

3.3.2 Viterbi Algorithm

We have implemented Bigram, Trigram and a heuristic improved Trigram Model. Here we explain the Trigram Model. All the models are well explained in the weekly reports submitted at btp21@10.6.0.90 in the directory btp21.
We use an efficient technique from dynamic programming to find the most probable tag sequence for given HMM, often termed as Viterbi Algorithm. The input to the algorithm is a sentence x(1) . . . x(n). Given this sentence, for any k in [1, n], for any sequence y(1) . . . y(k) such that y(i) ∈ K for i = 1 . . . k we define the following function

$$r(y_1 \dots y_k) = \prod_{i=1}^{k} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^{k} e(x_i | y_i)$$

$$\begin{aligned} p(x_1 \dots x_n, y_1 \dots y_{n+1}) &= r(y_1 \dots y_n) \times q(y_{n+1} | y_{n-1}, y_n) \\ &= r(y_1 \dots y_n) \times q(\text{STOP} | y_{n-1}, y_n) \end{aligned}$$

Further, for any any k ∈ {1 . . . n}, for any u ∈ K, v ∈ K, let S(k, u, v) to be the set of sequences y(1) . . . y(k) such that y(k−1) = u, y(k) = v, and y(i) ∈ K for i = 1 . . . k. Thus S(k, u, v) is the set of all tag sequences of length k, which end in the tag bigram (u, v). We also define

$$\pi(k, u, v) = \max_{\langle y_1 \dots y_k \rangle \in S(k,u,v)} r(y_1 \dots y_k)$$

Recursive definition:

For any k ∈ {1 . . . n}, for any u ∈ K and v ∈ K, we can use the following recursive definition

$$\pi(k, u, v) = \max_{w \in \mathcal{K}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$

π(k, u, v) is the highest probability for any sequence y(1) . . . y(k) ending in the bigram (u, v). Any such sequence must have y(k−2) = w for some state w. The highest probability for any sequence of length k − 1 ending in the bigram (w, u) is π(k − 1, w, u). Thus justified.

We now propose the following and complete the discussion.

$$\max_{y_1 \cdots y_{n+1}} p(x_1 \ldots x_n, y_1 \ldots y_{n+1}) = \max_{u \in \mathcal{K}, v \in \mathcal{K}} (\pi(n, u, v) \times q(\mathit{STOP}|u, v))$$

Algorithm Overview:

Input: a sentence $x_1 \ldots x_n$, parameters $q(s|u, v)$ and $e(x|s)$.
Initialization: Set $\pi(0, *, *) = 1$, and $\pi(0, u, v) = 0$ for all (u, v) such that $u \neq *$ or $v \neq *$.
Algorithm:

- For $k = 1 \ldots n$,
  - For $u \in \mathcal{K}$, $v \in \mathcal{K}$,

$$\pi(k, u, v) = \max_{w \in \mathcal{K}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v))$$

- **Return** $\max_{u \in \mathcal{K}, v \in \mathcal{K}} (\pi(n, u, v) \times q(\text{STOP}|u, v))$

Viterbi Algorithm with backpointers:

Input: a sentence $x_1 \ldots x_n$, parameters $q(s|u, v)$ and $e(x|s)$.
Initialization: Set $\pi(0, *, *) = 1$, and $\pi(0, u, v) = 0$ for all (u, v) such that $u \neq *$ or $v \neq *$.
Algorithm:

- For $k = 1 \ldots n$,
  - For $u \in \mathcal{K}$, $v \in \mathcal{K}$,

$$\begin{aligned} \pi(k, u, v) &= \max_{w \in \mathcal{K}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v)) \\ bp(k, u, v) &= \arg\max_{w \in \mathcal{K}} (\pi(k-1, w, u) \times q(v|w, u) \times e(x_k|v)) \end{aligned}$$

- Set $(y_{n-1}, y_n) = \arg\max_{(u,v)} (\pi(n, u, v) \times q(\text{STOP}|u, v))$
- For $k = (n-2) \ldots 1$,

$$y_k = bp(k+2, y_{k+1}, y_{k+2})$$

- **Return** the tag sequence $y_1 \ldots y_n$

3.3.3 Analysis of Algorithm

Brute-force Approach	Viterbi Algorithm
● Time Complexity: O(|k|^n) ● Exponential with respect to n ● Slower for larger n ● Considers all possible sequences	● Time Complexity: O(n*|k^3|) (Trigram) ● Linear with respect to n, Cubic with k ● Faster for larger n ● Uses dynamic programming

3.4 Smoothing

3.4.1 Data Sparsity:

Problem:
When a new, unknown, ambiguous word ‘w’ arrives, we never have seen any tag with this word while training. Hence, for all tags t ∈ T: **P(w | t) = 0.**

This is a major issue for building efficient taggers. Hindi has a large vocabulary. Moreover, words exist in more than one forms by changing prefixes and suffixes. This creates more ambiguous words along with the set of unknown words. Therefore, building a state-of-art POS tagger ensures proper handling these set of words.

One solution is to define set of rules for these words and combine Rule-based approach here. But again, as discussed above in previous sections, this method is not suitable for Hindi in th current scenario.
So we move towards another technique, called ‘Smoothing’. Various smoothing techniques exist till date, many incorporated in popular English and other language taggers. For example, Good Turing Smoothing, Kneser-Ney Smoothing, etc.

We incorporate Laplace Smoothing and Offset Smoothing methods in our approach.

3.4.2 Laplace Smoothing

‘Laplace Smoothing’, often referred as naive approach or one count smoothing, is an additive technique used for smoothing of categorical data.

Calculation of HMM parameters is done as follows:

$$q(s|u,v) = \frac{c(u,v,s)}{c(u,v)}$$

$$e(x|s) = \frac{c(s \rightsquigarrow x)}{c(s)}$$

The following are values that can go wrong here:

- c(u, v, s) is 0
- c(u, v) is 0
- We get an unknown word in the test sentence, and we don't have any training tags associated with it.

To solve all these, Laplace Smoothing adds some changes in these HMM parameter calculations which become as follows:

$$q(s|u,v) = \frac{c(u,v,s)+\lambda}{c(u,v)+\lambda V}$$

$$e(x|s) = \frac{c(s \rightsquigarrow x)+\lambda}{c(s)+\lambda V}$$

Here V is the total number of tags in our corpus and λ is basically a real value between 0 and 1. It acts like a discounting factor. A λ = 1 value would give us too much of a redistribution of values of probabilities.

3.4.3 Offset Smoothing

This is used to improve the accuracy of every Hidden Markov Model in our algorithms to provide a kind of offset or base so that unknown emission probabilities have a kind of reasonable probability in the negative order of 10 according to the words in the corpus.

The analysis for same for finding appropriate value for offsets is given further in Graphical Analysis part of the report.

3.5 Decision Trees

A decision tree is a simple flowchart that selects labels for input values. This flowchart consists of decision nodes, which check feature values, and leaf nodes, which assign labels. To choose the label for an input value, we begin at the flowchart's initial decision node, known as its root node. This node contains a condition that checks one of the input value's features, and selects a branch based on that feature's value. Following the branch that describes our input value, we arrive at a new decision node, with a new condition on the input value's features. We continue following the branch selected by each node's condition, until we arrive at a leaf node which provides a label for the input value.

However, decision trees also have a few disadvantages. One problem is that, since each branch in the decision tree splits the training data, the amount of training data available to train nodes lower in the tree can become quite small. As a result, these lower decision nodes may overfit the training set, learning patterns that reflect idiosyncrasies of the training set rather than linguistically significant patterns in the underlying problem. One solution to this problem is to stop dividing nodes once the amount of training data becomes too small. Another solution is to grow a full decision tree, but then to prune decision nodes that do not improve performance on a dev-test.

A second problem with decision trees is that they force features to be checked in a specific order, even when features may act relatively independently of one another. For example, when classifying documents into topics (such as sports, automotive, or murder mystery), features such as hasword(football) are highly indicative of a specific label, regardless of what other the feature values are. Since there is limited space near the top of the decision tree, most of these features will need to be repeated on many different branches in the tree. And since the number of branches increases exponentially as we go down the tree, the amount of repetition can be very large.

A related problem is that decision trees are not good at making use of features that are weak predictors of the correct label. Since these features make relatively small incremental improvements, they tend to occur very low in the decision tree. But by the time the decision tree learner has descended far enough to use these features, there is not enough training data left to reliably determine what effect they should have. If we could instead look at the effect of these features across the entire training set, then we might be able to make some conclusions about how they should affect the choice of label.

4. Results and Analysis

How We Improved Our Trigram Hidden Markov Model

We expected a better output than bigram HMM on trigram HMM, but we came up with an issue, it being that of the 22 * 22 * 22 = 10648 possible trigrams, only about 4500 were getting values from the corpus and the rest 6000 were dependent on only smoothing for the probability values, even though we have a corpus which is very large. The same problem does not occur in bigram HMM because of the 22 * 22 = 484 bigrams, about 80-90% of the bigrams were being filled from the corpus and the algorithm outputs result with a high accuracy. So, to improve the trigram HMM, we split the trigram into 2 bigrams as a heuristic (u,v,w to u,v and v,w) and were able to get significantly better results, with the accuracy getting better by almost 3.1% to given an overall accuracy of 88.07%.

How We Improved Our Decision Tree Classification Model

We first trained our Decision Tree Classification model using features related to the neighbouring words, last words, first words, numerics etc. With these kinds of features our model was reporting an accuracy of about 78%. After some analysis and reading through research papers we got to know that because Hindi is a language which is morphologically very strong, so we had to incur some more features regarding the same. So, we added 6 more features to our model relating to the prefixes and suffixes and improved the accuracy of our model by almost 9%. We also got to know that while English and European languages could use a machine learning approach to get good results, Hindi couldn’t achieve the same. So, we trained a Decision Tree model for English and compared it with our Hindi based model. Even on a smaller dataset than Hindi, English was trained significantly better than Hindi by about 3.5%, with an accuracy of about 89.5% whereas our improved Hindi based model with morphological features had an accuracy of around 87%. Also, this supported the claims of the research papers that we read during our project.

4.1 Output Figures

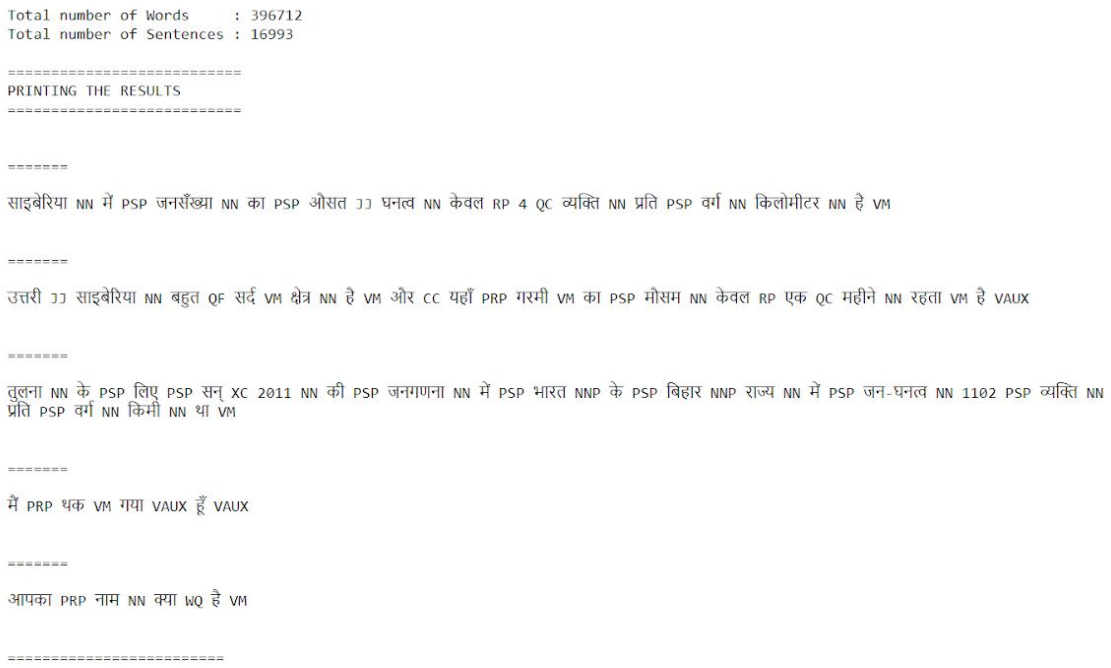


Figure 1: Sample output for Bigram HMM

Figure 2: Sample output for Trigram HMM

Circle marks show differences with Bigram HMM

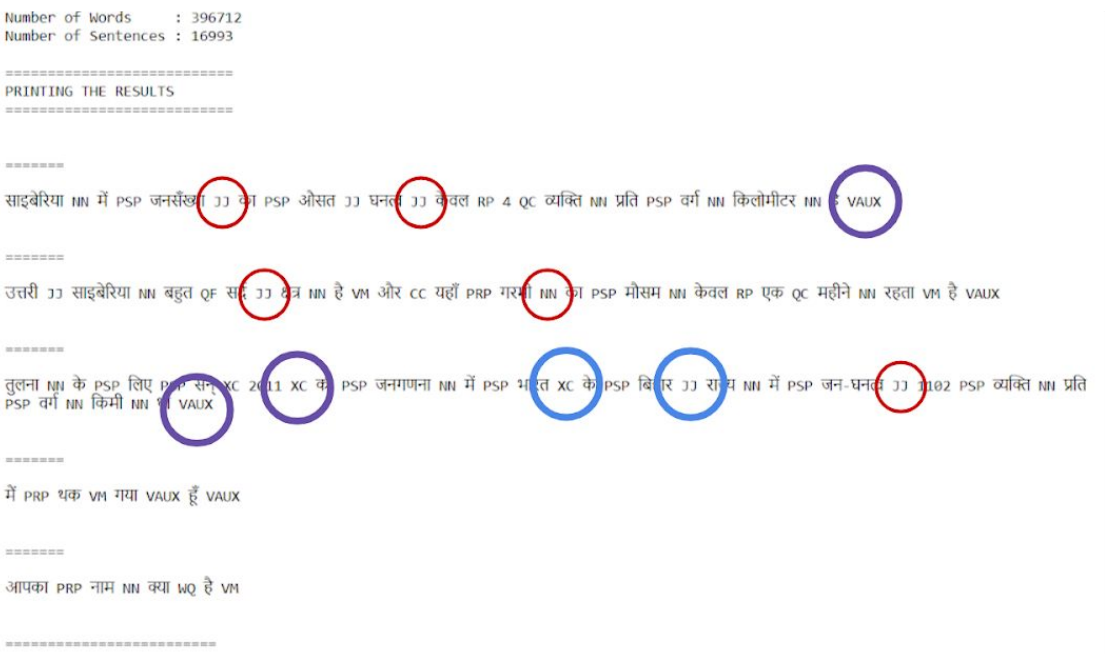


Figure 3: Sample output for Trigram HMM

Circle marks show differences with Bigram and Trigram HMM

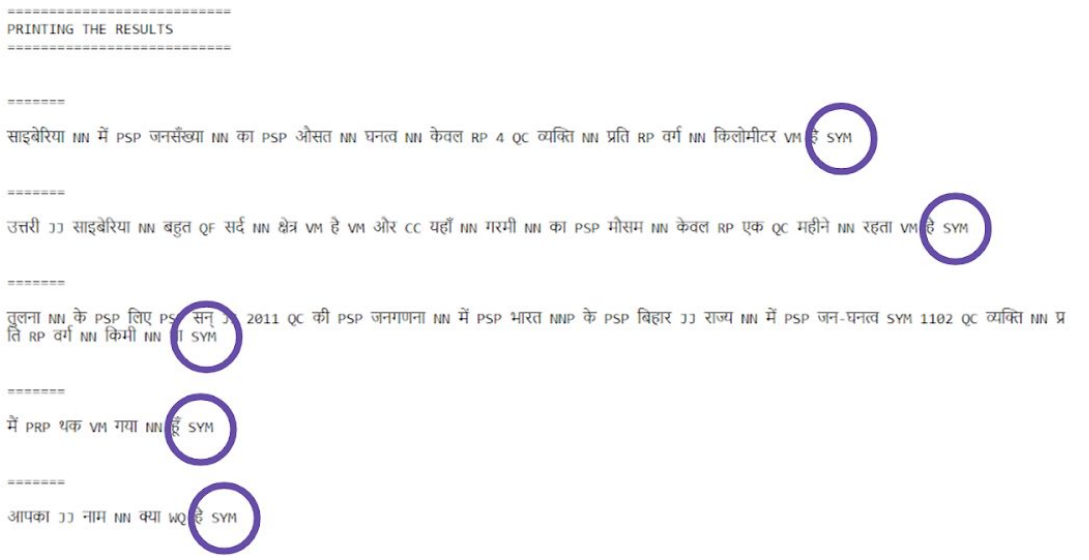


Figure 4: Sample output for Decision Tree Model

Circle marks show incorrect Overfitting for 'Full-Stop' (end-of-line character) in Hindi, which should not be here because the last-to-end of line characters are usually VAUX, VM, etc. We deliberately did not put any Hindi full stop at the end of input lines.

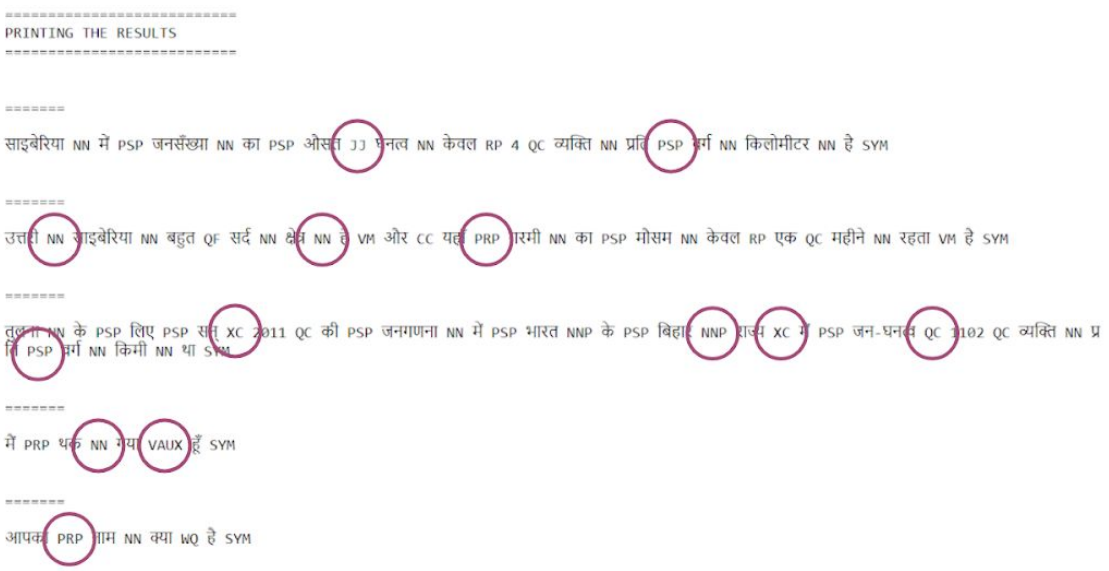
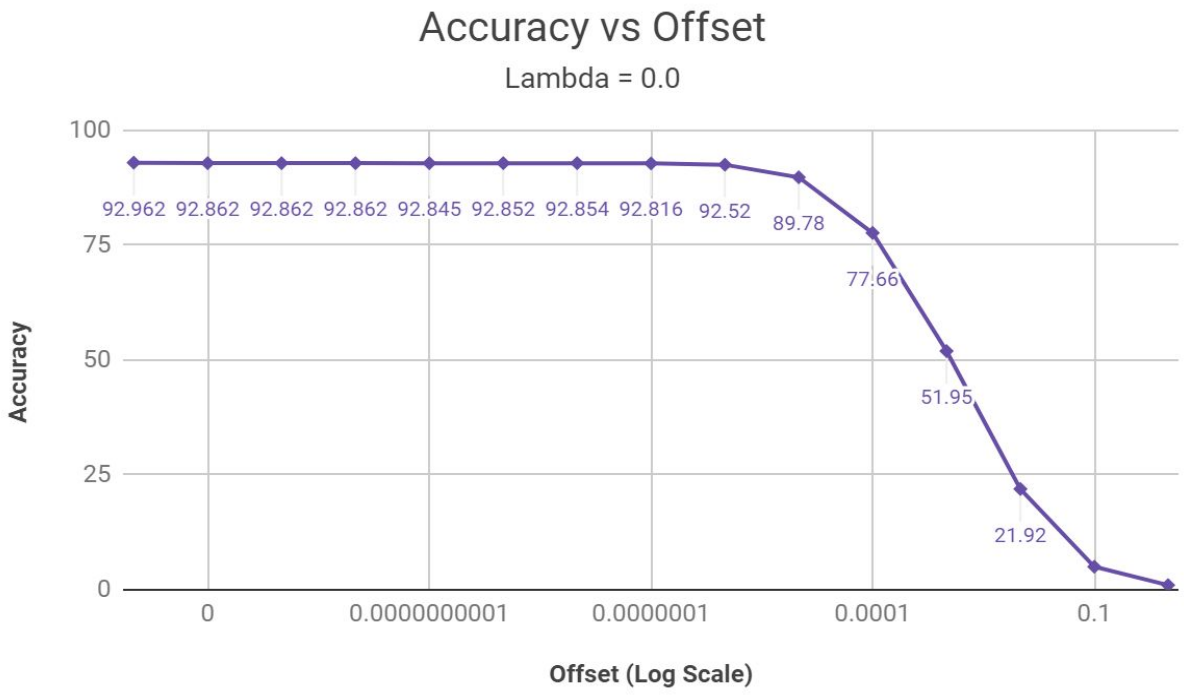


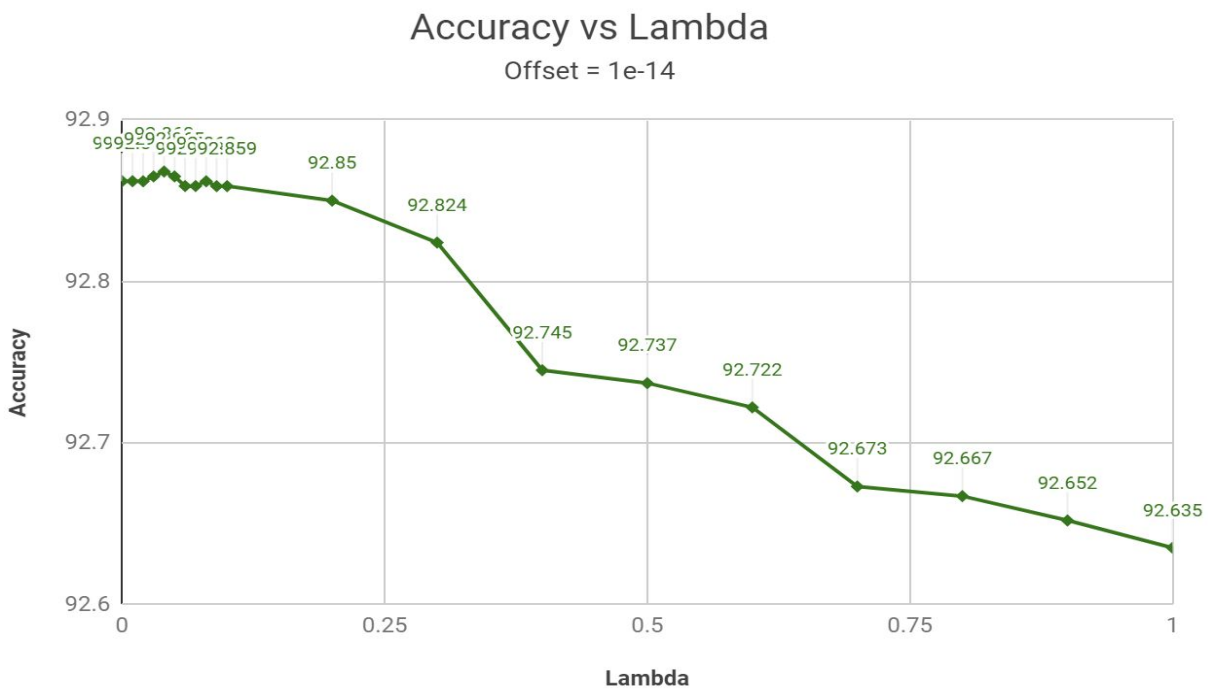
Figure 5: Sample output for Improved Decision Tree Model

Circle marks show differences with Decision Tree Model, with the majority being corrections of the previous decision tree model.

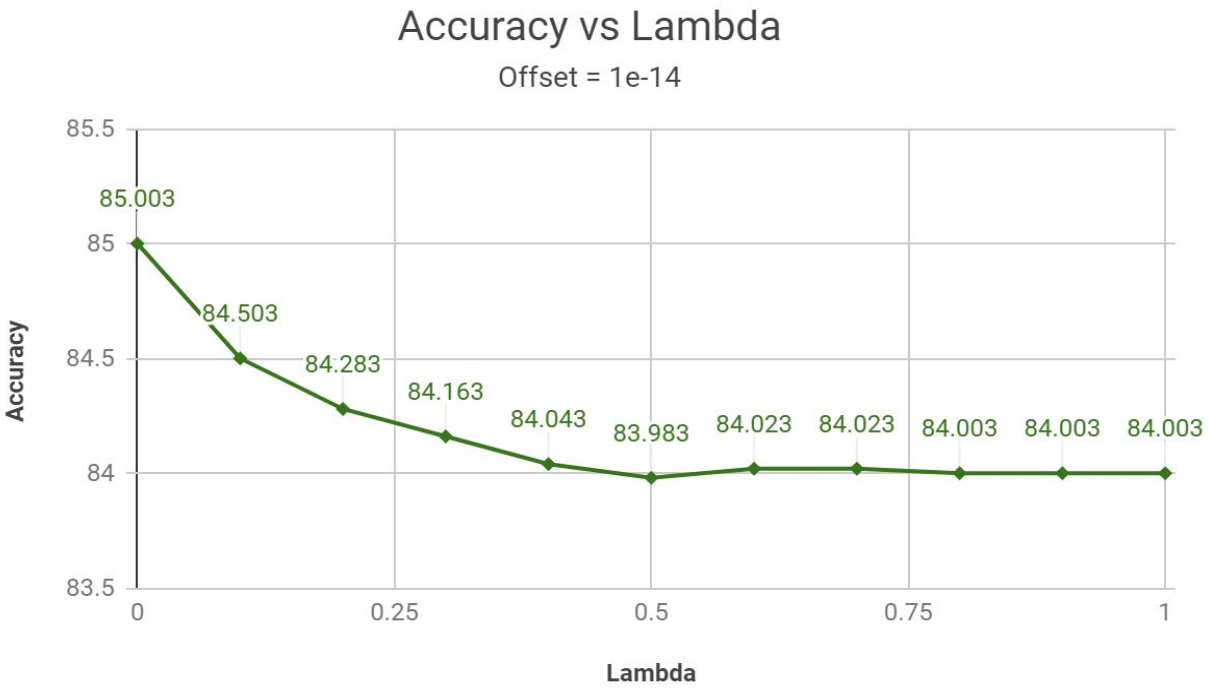
4.2 Graphical Analysis



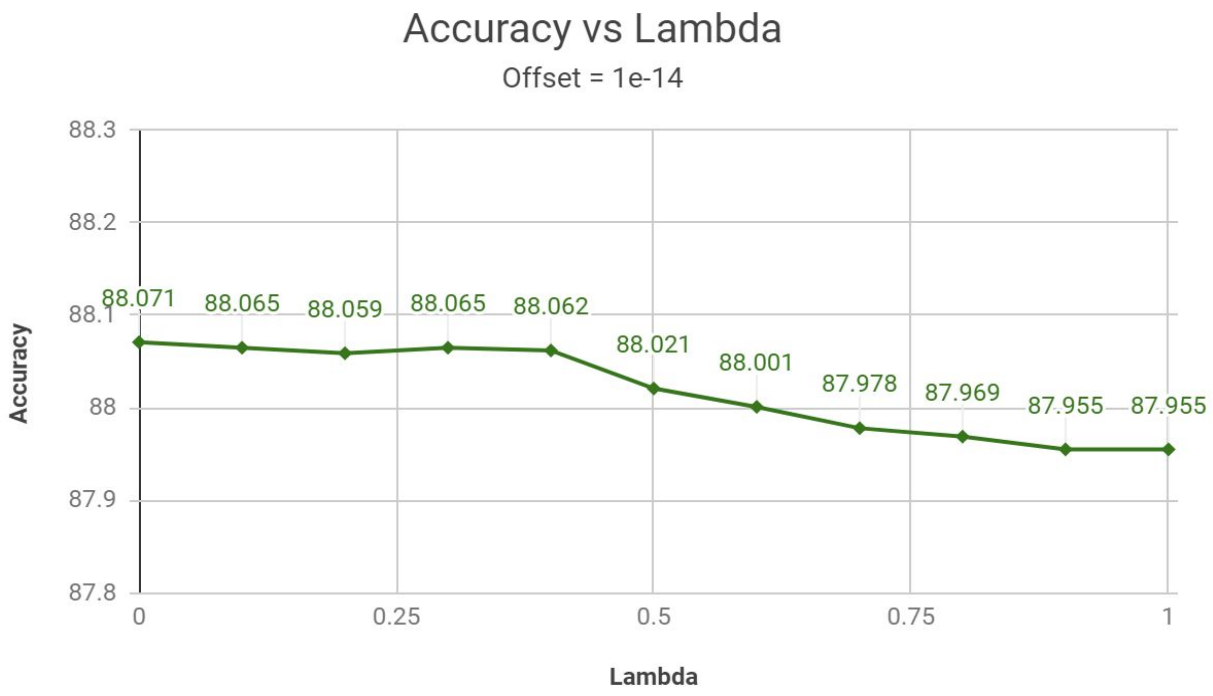
Graph 1: Accuracy vs Offset for Bigram HMM (Log scaled)



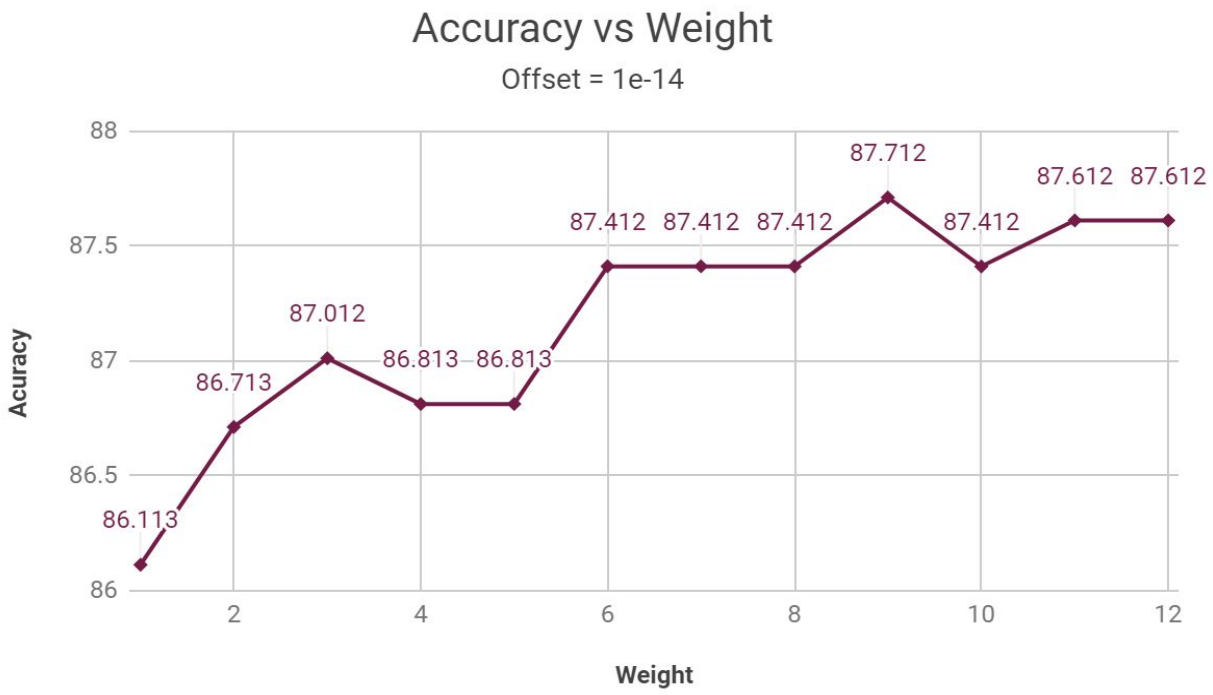
Graph 2: Accuracy vs Lambda for Bigram HMM



Graph 3: Accuracy vs Lambda for Trigram HMM



Graph 4: Accuracy vs Lambda for Improved Trigram HMM



Graph 5: Accuracy vs Weight for Improved Trigram HMM

4.3 Accuracy Comparison

Bigram HMM vs Trigram HMM

```
CORRECT TAGS : 29206  ========  TOTAL TAGS : 34369
CORRECT TAGS : 29207  ========  TOTAL TAGS : 34370
CORRECT TAGS : 29207  ========  TOTAL TAGS : 34371
CORRECT TAGS : 29208  ========  TOTAL TAGS : 34372
CORRECT TAGS : 29209  ========  TOTAL TAGS : 34373
CORRECT TAGS : 29210  ========  TOTAL TAGS : 34374
CORRECT TAGS : 29211  ========  TOTAL TAGS : 34375
CORRECT TAGS : 29212  ========  TOTAL TAGS : 34376
CORRECT TAGS : 29213  ========  TOTAL TAGS : 34377
CORRECT TAGS : 29214  ========  TOTAL TAGS : 34378
CORRECT TAGS : 29215  ========  TOTAL TAGS : 34379
CORRECT TAGS : 29216  ========  TOTAL TAGS : 34380
============================
ACCURACY : 84.9796393252 % Offset : 1e-14 Lambda : 0.0
============================
```

```
CORRECT TAGS : 31916  ========  TOTAL TAGS : 34368
CORRECT TAGS : 31917  ========  TOTAL TAGS : 34369
CORRECT TAGS : 31918  ========  TOTAL TAGS : 34370
CORRECT TAGS : 31919  ========  TOTAL TAGS : 34371
CORRECT TAGS : 31920  ========  TOTAL TAGS : 34372
CORRECT TAGS : 31921  ========  TOTAL TAGS : 34373
CORRECT TAGS : 31922  ========  TOTAL TAGS : 34374
CORRECT TAGS : 31923  ========  TOTAL TAGS : 34375
CORRECT TAGS : 31924  ========  TOTAL TAGS : 34376
CORRECT TAGS : 31925  ========  TOTAL TAGS : 34377
CORRECT TAGS : 31926  ========  TOTAL TAGS : 34378
CORRECT TAGS : 31927  ========  TOTAL TAGS : 34379
CORRECT TAGS : 31928  ========  TOTAL TAGS : 34380
============================
ACCURACY : 92.8679464805 % Offset : 1e-14 Lambda : 0.04
============================
```

Trigram HMM vs Improved Trigram HMM

```
CORRECT TAGS : 29207  ========  TOTAL TAGS : 34370
CORRECT TAGS : 29207  ========  TOTAL TAGS : 34371
CORRECT TAGS : 29208  ========  TOTAL TAGS : 34372
CORRECT TAGS : 29209  ========  TOTAL TAGS : 34373
CORRECT TAGS : 29210  ========  TOTAL TAGS : 34374
CORRECT TAGS : 29211  ========  TOTAL TAGS : 34375
CORRECT TAGS : 29212  ========  TOTAL TAGS : 34376
CORRECT TAGS : 29213  ========  TOTAL TAGS : 34377
CORRECT TAGS : 29214  ========  TOTAL TAGS : 34378
CORRECT TAGS : 29215  ========  TOTAL TAGS : 34379
CORRECT TAGS : 29216  ========  TOTAL TAGS : 34380
============================
ACCURACY : 84.9796393252 % Offset : 1e-14 Lambda : 0.0
============================
```

```
CORRECT TAGS : 30451  ========  TOTAL TAGS : 34577
CORRECT TAGS : 30452  ========  TOTAL TAGS : 34578
CORRECT TAGS : 30453  ========  TOTAL TAGS : 34579
CORRECT TAGS : 30454  ========  TOTAL TAGS : 34580
CORRECT TAGS : 30455  ========  TOTAL TAGS : 34581
CORRECT TAGS : 30456  ========  TOTAL TAGS : 34582
CORRECT TAGS : 30457  ========  TOTAL TAGS : 34583
CORRECT TAGS : 30458  ========  TOTAL TAGS : 34584
CORRECT TAGS : 30459  ========  TOTAL TAGS : 34585
CORRECT TAGS : 30460  ========  TOTAL TAGS : 34586
CORRECT TAGS : 30461  ========  TOTAL TAGS : 34587
CORRECT TAGS : 30462  ========  TOTAL TAGS : 34588
============================
ACCURACY : 88.0710072858 % Offset : 1e-14 Lambda : 0.0
============================
```

Decision Tree vs Improved Decision Tree (which has more morphological features)

```
Number of tagged sentences in dataset : 6993
Number of tagged words in dataset     : 163433
The number of training sentences are  : 5244
The number of test sentences are      : 1749

Please wait...Training the model.

Training completed.

====================

ACCURACY : 78.06219234001681

====================
```

```
Number of tagged sentences in dataset : 6993
Number of tagged words in dataset     : 163433
The number of training sentences are  : 5244
The number of test sentences are      : 1749

Please wait...Training the model.

Training completed.

====================

ACCURACY : 86.69948373154041

====================
```

Conclusion

We have successfully implemented a Part-of-Speech Tagger for Hindi Language. We have constructed Hidden Markov Model and implemented Viterbi Algorithm for determining tag sequences. We also implemented POS Tagger using Decision Tree Classification Model.

We achieved following accuracies:

Hindi Bigram Hidden Markov Model = 92.867%
Hindi Trigram Hidden Markov Model = 84.979%
Hindi Improved Trigram Hidden Markov Model = 88.071%
Hindi Improved Decision Tree Classification Model = 86.699%

Future Work

We can extend this approach to construct a Hybrid Part-of-Speech Tagger for Hindi using both, Rule based Approach and Stochastic Methods.

We can implement different, efficient smoothing techniques, like Good Turing Smoothing, Katz Smoothing etc. and increase accuracy even further.

We can improve the dataset by extending the size of the corpus and increase frequencies for individual tags. Also we can increase different possibilities of trigrams, because earlier, of the 10648 possibilities, only around 4500 were available in the dataset and so we had to apply smoothing to the rest 6000 trigrams.

Code Snippets

Here we give some snippets of our code. The code is also available in the directory btp21 at btp21@10.6.0.90.

```
    # Finally getting the answer of the viterbi algorithm, element[1] is the probability of the final iteration possibilities
    viterbiAnswer = max(maxPossibilitiesList, key = lambda element:element[1])

    # Getting the tags associated with the viterbiAnswer by referring to the backtrackingDP
    tagsAssigned = backtrackingDict[viterbiAnswer[0]][1:]

    # Removing the '*' appended in the starting of algorithm, not required anymore
    sentence = sentence[1:]

    # Returning the answer
    return zip(sentence, tagsAssigned)


def calculateTransitionProbabilities(trainFile, Lambda):

    # Declaring the variables and default dicts required
    transitionProbabilityDict = defaultdict(int)
    bigramTransitionCountDict = defaultdict(int)
    unigramTransitionCountDict = defaultdict(int)
    numberOfWords = 0
    numberOfSentences = 0
    allTags = set([])

    # Iterating through the lines of the input file
    for line in trainFile.readlines():

        # Getting tokens from each such line
        tokens = line.split()

        # Initializing a list for the tags observed in the line
        tags = []

        # For each token in that line
        for token in tokens:

            # Extracting the tag by splitting and stripping according to the file
            tag = token.split('|')[2].split('.')[0].strip(':?').strip()

            # Giving exact tags in training data a common parent tag for less complexity
            # and more accuracy
            if (tag == 'I-NP' or tag == 'B-NP' or tag == 'O'):
                tag = 'NN'

            allTags = allTags | set([tag])

            # Appending the tag to the list of tags
            tags.append(tag)

        # Incrementing the total number of words by the required amount
        numberOfWords += len(tags)

        # If the line read is not a blank line
        if tags != []:
```

```
            # Calculating the transition probability
            transitionProbability = (bigramTransitionCountDict[bigram] + Lambda ) / (unigramTransitionCountDict[uUnigram] + (Lambda
* 22))

            # Storing the transition probability in the dict
            transitionProbabilityDict[keyVgivenU] = transitionProbability

    # Printing the total number of words and sentences
    print("Total number of Words     : " + str(numberOfWords))
    print("Total number of Sentences : " + str(numberOfSentences))

    # Returning probability dict and tags set
    return transitionProbabilityDict, allTags


def calculateEmissionProbabilities(trainFile, Lambda):

    # Declaring the variables and default dicts required
    # 0.000000001 as offset for smoothing
    emissionProbabilityDict = defaultdict(lambda : 0.000000001)
    emissionCountDict = defaultdict(int)
    separateTagCountDict = defaultdict(int)

    # Iterating through the lines of the input file
    for line in trainFile.readlines():

        # Getting tokens form each such line
        tokens = line.split()

        # Initializing a list for the tags observed in the line
        tags = []

        # For each token in that line
        for token in tokens:

                # Extracting the word by splitting and stripping according to the file
                word  = token.split('|')[0].strip()
                # Extracting the tag by splitting and stripping according to the file
                tag = token.split('|')[2].split('.')[0].strip(':?').strip()

                # Giving exact tags in training data a common parent tag for less complexity
                # and more accuracy
                if (tag == 'I-NP' or tag == 'B-NP' or tag == 'O'):
                    tag = 'NN'

                # Appending the tag to the list of tags
                tags.append(tag)

                # Calculating the number of times the tag and word appear together
                emissionCountDict[tag + '|' + word] += 1

                # Calculating the number of times that tag appears in general
                separateTagCountDict[tag] += 1
```

```
    print("\n==========================\nPRINTING THE RESULTS\n==========================\n")
    # Iterating through each of the sentences
    for sentence in sentencesList:

        # Calling the algorithm on the sentence
        predictedWordsAndTags = bigramHMMViterbiAlgorithm(sentence, allTags, emissionProbabilityDict, transitionProbabilityDict)
        print("\n======\n")
        # Printing the result of the algorithm
        for (word, tag) in predictedWordsAndTags:
            print word, tag,
        print("\n")
    print("\n========================\n")

def bigramHMMViterbiAlgorithm(sentence, allTags, emissionProbabilityDict, transitionProbabilityDict):

    # Appending a '*' required for the initial bigram in the algorithm(base case)
    sentence = [ u'*'] + sentence

    # Declaring the variables required
    initialLength = len(sentence)
    backtrackingDict = defaultdict(list)
    dpDict = {}
    tagsAssigned = []

    # Initializing the base cases
    dpDict[0, '*'] = 1
    for u in ([ u'*' ] + list(allTags)):
            if u != u'*':
                dpDict[0, u] = 1

    # Iterating through each word in the sentence from starting in terms of length
    for k in range(1, initialLength):
        # For all possibilities of (u, v) tags
        for v in allTags:

            # List for possibilities for getting maximum probability
            possibilities = []

            # For all possibilities of (u, v) tags
            for u in allTags:

                # Viterbi algorithm formula
                possibility = dpDict[k - 1 , u] * transitionProbabilityDict[(v, u)] * (emissionProbabilityDict[sentence[k] + '|'
+ v])

                # Appending the possibility to the list of possibilities
                possibilities.append((possibility, u))

            # Getting the maximum probability from the list of possibilities, element[0] is the probability of the possibility
            maxUVgivenK = max(possibilities, key = lambda element : element[0])

            # Storing the answer in the DP dict
            dpDict[k, v] = maxUVgivenK[0]

            # Getting the element to be backtracked and storing it in the backtracking Dict
```

```
    for sentence in sentencesList:

        # Calling the algorithm on the sentence
        predictedWordsAndTags = trigramHMMViterbiAlgorithm(sentence, allTags, emissionProbabilityDict, transitionProbabilityDict
)
        print("\n=======\n")
        # Printing the result of the algorithm
        for (word, tag) in predictedWordsAndTags:
            print word, tag,
        print("\n")
    print("\n==========================\n")

def trigramHMMViterbiAlgorithm(sentence, allTags, emissionProbabilityDict, transitionProbabilityDict):

    # Appending two '*'s required for the initial trigram in the algorithm(base case)
    sentence = [ u'*', u'*' ] + sentence

    # Declaring the variables required
    initialLength = len(sentence)
    backtrackingDict = defaultdict(list)
    dpDict = {}
    tagsAssigned = []

    # Initializing the base cases
    dpDict[0, '*', '*'] = 1
    for u in ([ u'*' ] + list(allTags)):
        for v in ([ u'*' ] + list(allTags)):
            if u != u'*' or v != u'*':
                dpDict[0, u, v] = 1

    # Iterating through each word in the sentence from starting in terms of length
    for k in range(1, initialLength):
        # For all possibilities of (w, u, v) tags
        for v in allTags:
            # For all possibilities of (w, u, v) tags
            for u in allTags:

                # List for possibilities for getting maximum probability
                possibilities = []

                # For all possibilities of (w, u, v) tags
                for w in allTags:

                    # Viterbi algorithm formula
                    possibility = dpDict[k - 1 , w, u] * transitionProbabilityDict[(v, w, u)] * (emissionProbabilityDict[sentenc
e[k] + '|' + v])

                    # Appending the possibility to the list of possibilities
                    possibilities.append((possibility, w))

                # Getting the maximum probability from the list of possibilities, element[0] is the probability of the possibili
ty
                maxWUVgivenK = max(possibilities, key = lambda element:element[0])

                # Storing the answer in the DP dict
                dpDict[k, u, v] = maxWUVgivenK[0]

                # Getting the element to be backtracked and storing it in the backtracking Dict
                backtrackW = maxWUVgivenK[1]
                backtrackingDict[u, v] += [backtrackW]

                # Additionally storing (u, v) of (w, u, v) tags in the last iteration
```

[1]:
```
# -*- coding: utf-8 -*-

from __future__ import division
import codecs
from collections import defaultdict

def main():

    # Opening the input training file
    trainFile = codecs.open("trainDataHindi.txt", mode = "r", encoding = "utf-8")

    # Getting the emission probabilities
    emissionProbabilityDict = calculateEmissionProbabilities(trainFile, 0.0)

    # Opening the input training file
    trainFile = codecs.open("trainDataHindi.txt", mode = "r", encoding = "utf-8")

    # Getting the transition probabilities and all the tags (states)
    transitionProbabilityDict, allTags = calculateTransitionProbabilities(trainFile, 0.0)

    # Opening the input file
    inputSentences = codecs.open("input.txt", mode = "r", encoding = "utf-8")

    # Initializing list for storing input sentences
    sentencesList = []

    # Reading the input file and storing the sentences
    for line in inputSentences:
        sentence = []
        tokens = line.split()
        for token in tokens:
            word = token.split('|')[0].strip()
            sentence.append(token)

        # Appending the sentence in the list of sentences
        sentencesList.append(sentence)

    print("\n==========================\nPRINTING THE RESULTS\n==========================\n")
    # Iterating through each of the sentences
    for sentence in sentencesList:

        # Calling the algorithm on the sentence
        predictedWordsAndTags = trigramHMMViterbiAlgorithm(sentence, allTags, emissionProbabilityDict, transitionProbabilityDict
)
        print("\n=======\n")
        # Printing the result of the algorithm
        for (word, tag) in predictedWordsAndTags:
            print word, tag,
        print("\n")
    print("\n========================\n")


def trigramHMMViterbiAlgorithm(sentence, allTags, emissionProbabilityDict, transitionProbabilityDict):

    # Appending two '*'s required for the initial trigram in the algorithm(base case)
```

```
        # Printing the accuracy result
        print("============================")
        print("ACCURACY : " + str(accuracy) + " %")
        print("============================")
        time.sleep(1)

def testTrigramHMMViterbiAlgorithm(testFile, allTags, emissionProbabilityDict, transitionProbabilityDict):

    # Initializing the variables required for calculating accuracy
    totalTags = 0
    correctTags = 0

    # Reading the file line by line
    for line in testFile.readlines():

        # Getting tokens from each such line
        tokens = line.split()

        # Initializing a list for the words observed in the line
        sentence = []

        # Initializing a list for the tags observed in the line
        tags = []

        for token in tokens:

            # Extracting the word by splitting and stripping according to the file
            word = token.split('|')[0].strip()
            # Extracting the tag by splitting and stripping according to the file
            tag = token.split('|')[2].split('.')[0].strip(':?').strip()

            # Giving exact tags in training data a common parent tag for less complexity
            # and more accuracy
            if (tag == 'I-NP' or tag == 'B-NP' or tag == 'O'):
                tag = 'NN'

            # Appending the word to the list of word
            sentence.append(word)
            # Appending the tag to the list of tags
            tags.append(tag)

        # If the line read is not a blank line
        if sentence != []:

            # Zipping to get a list of (word, tag) tuple elements format
            actualWordsAndTags = zip(sentence, tags)

            try:
                # Calling the function on the sentence
                predictedWordsAndTags = trigramHMMViterbiAlgorithm(sentence, allTags, emissionProbabilityDict, transitionProbabi
lityDict)

                # Iterating through to compare the predicted and actual tags
                for i in range(len(sentence)):
```

```
        testFile = codecs.open("testDataHindi.txt", mode = "r", encoding = "utf-8")

        # Calling the test function to get the accuracy of algorithm on the test data file
        accuracy = testBigramHMMViterbiAlgorithm(testFile, allTags, emissionProbabilityDict, transitionProbabilityDict, Lambda)

        # Printing the accuracy result
        print("============================")
        print("ACCURACY : " + str(accuracy) + " %")
        print("============================")
        time.sleep(1)

def testBigramHMMViterbiAlgorithm(testFile, allTags, emissionProbabilityDict, transitionProbabilityDict, Lambda):

    # Initializing the variables required for calculating accuracy
    totalTags = 0
    correctTags = 0

    # Reading the file line by line
    for line in testFile.readlines():

        # Getting tokens from each such line
        tokens = line.split()

        # Initializing a list for the words observed in the line
        sentence = []

        # Initializing a list for the tags observed in the line
        tags = []

        for token in tokens:

            # Extracting the word by splitting and stripping according to the file
            word = token.split('|')[0].strip()
            # Extracting the tag by splitting and stripping according to the file
            tag = token.split('|')[2].split('.')[0].strip(':?').strip()

            # Giving exact tags in training data a common parent tag for less complexity
            # and more accuracy
            if (tag == 'I-NP' or tag == 'B-NP' or tag == 'O'):
                tag = 'NN'

            # Appending the word to the list of words
            sentence.append(word)
            # Appending the tag to the list of tags
            tags.append(tag)

        # If the line read is not a blank line
        if sentence != []:

            # Zipping to get a list of (word, tag) tuple elements format
            actualWordsAndTags = zip(sentence, tags)

            try:
                # Calling the function on the sentence
                predictedWordsAndTags = bigramHMMViterbiAlgorithm(sentence, allTags, emissionProbabilityDict, transitionProbabil
```

```
[1]: # -*- coding: utf-8 -*-

# Importing the libraries required
import nltk, codecs
from sklearn.tree import DecisionTreeClassifier
from sklearn.feature_extraction import DictVectorizer
from sklearn.pipeline import Pipeline

# TRAINING DECISION TREE POS TAGGER IN HINDI USING THE ABOVE LIBRARIES

# Initializing the classifier to be used
classifier = Pipeline([
    ('vectorizer', DictVectorizer(sparse = False)),
    ('classifier', DecisionTreeClassifier(criterion = 'entropy'))
])

def main():

    # Opening training file
    trainFile = codecs.open("trainDataHindi.txt", mode = "r", encoding = "utf-8")

    # Getting the nltk corpus treebank consisting of tagged sentences
    taggedSentences = getTaggedSentences(trainFile)[:-10000]

    # Printing the number of tagged sentences and words in the same
    print("Number of tagged sentences in dataset : " + str(len(taggedSentences)))
    numberOfWords = 0
    for sentence in taggedSentences:
        numberOfWords += len(sentence)

    print("Number of tagged words in dataset     : " + str(numberOfWords))

    # # Printing an example illustrating the features used in the model
    # exampleFeatures = features(['मेरा', 'नाम', 'शाश्वत', 'कपूरिया', 'है', '2011'], 2)
    # for key in exampleFeatures:
    #     print key,
    #     print exampleFeatures[key]

    # Training 75% of tagged sentences as it is an ideal partition
    cutoff = int(.75 * len(taggedSentences))

    # Splitting the same to learn from 75% and then test on 25% of data
    trainingSentences = taggedSentences[:cutoff]
    testSentences = taggedSentences[cutoff:]

    # Printing the number of tagged sentences and test sentences
    print("The number of training sentences are  : " + str(len(trainingSentences)))
    print("The number of test sentences are      : " + str(len(testSentences)))

    # Tranforming to dataset to use inbuilt classifier function to train the model
    X, y = transformToDataset(trainingSentences)

    print("\nPlease wait...Training the model.\n")

    # Training(Fitting) the model
```

```
    # Reading the input file and storing the sentences
    for line in inputSentences:
        sentence = []
        tokens = line.split()
        for token in tokens:
            word = token.split('|')[0].strip()
            sentence.append(token)

        # Appending the sentence in the list of sentences
        sentencesList.append(sentence)

    print("\n============================\nPRINTING THE RESULTS\n============================\n")
    # Iterating through each of the sentences
    for sentence in sentencesList:

        # Calling the algorithm on the sentence
        predictedWordsAndTags = posTag(sentence)
        print("\n=======\n")
        # Printing the result of the algorithm
        for (word, tag) in predictedWordsAndTags:
            print word, tag,
        print("\n")
    print("\n=========================\n")

def features(sentence, index):
    """Function to return the features to be used in the model. Index is the index of the specific word in the sentence."""

    # Returning the necessary features applied to the word, i.e., sentence[index]
    return {
        'word': sentence[index],
        'isFirst': index == 0,
        'isLast': index == len(sentence) - 1,
        'prefix1': sentence[index][0],
        'prefix2': sentence[index][:2],
        'prefix3': sentence[index][:3],
        'suffix1': sentence[index][-1],
        'suffix2': sentence[index][-2:],
        'suffix3': sentence[index][-3:],
        'previousWord': '' if index == 0 else sentence[index - 1],
        'nextWord': '' if index == len(sentence) - 1 else sentence[index + 1],
        'hasHyphen': '-' in sentence[index],
        'isNumeric': sentence[index].isdigit()
    }

def untag(taggedSentence):
    """Function to strip the tags from the sentences in our trained corpus and return the list of only words."""
    return [word for word, tag in taggedSentence]


def transformToDataset(taggedSentences):
    """Function to tranform the tagged sentences into dataset suitable to pass into inbuilt classifier function."""
    # X is the list of word specific the features and y is the corresponding tags
    X, y = [], []

    # Appending the corresponding values of features and tags to X and y respectively
```

```
    return X, y

def posTag(sentence):
    tags = classifier.predict([features(sentence, index) for index in range(len(sentence))])
    return zip(sentence, tags)


def getTaggedSentences(trainFile):
    """Function to get the tagged sentences in the input file. Output is the list of sentences
       with elements as word, tag tuples."""

    # Initializing all tags list as a set
    allTags = set([])

    # Initializing sentences list
    sentencesList = []

    # Iterating through the lines of the input file
    for line in trainFile.readlines():

        # Getting tokens from each such line
        tokens = line.split()

        sentence = []

        # Initializing a list for the tags observed in the line
        tags = []

        # For each token in that line
        for token in tokens:

            word = token.split('|')[0].strip()
            # Extracting the tag by splitting and stripping according to the file
            tag = token.split('|')[2].split('.')[0].strip(':?').strip()

            # Giving exact tags in training data a common parent tag for less complexity
            # and more accuracy
            if (tag == 'I-NP' or tag == 'B-NP' or tag == 'O'):
                tag = 'NN'

            # Appending (word, tag) tuple to the sentence
            sentence.append((word, tag))

            allTags = allTags | set([tag])

            # Appending the tag to the list of tags
            tags.append(tag)

        if len(sentence) > 0:
            # Appending the sentence to the list of sentences
            sentencesList.append(sentence)

    return sentencesList

if __name__ == "__main__":
    main()
```

References

Research Papers and references linked in the weekly reports and ref.json in the directory btp21 at btp21@10.6.0.90.

Dalal, Aniket, et al. "Hindi part-of-speech tagging and chunking: A maximum entropy approach." Proceeding of the NLPAI Machine Learning Competition (2006).

Sankaran Baskaran, “Hindi POS Tagging and Chunking”, In Proceedings of the NLPAI Machine Learning 2006 Competition.

Shrivastava, M., & Bhattacharyya, P. (2008, December). Hindi pos tagger using naive stemming: Harnessing morphological information without extensive linguistic knowledge. In International Conference on NLP (ICON08), Pune, India.

Avinesh PVS, Karthik G. 2007. Part Of Speech Tagging and Chunking using Conditional Random Fields and Transformation Based Learning. In Proceedings of IJCAI Workshop on ”Shallow Parsing for South Asian Languages”

http://www.cs.columbia.edu/~mcollins/courses/nlp2011

https://www.cs.bgu.ac.il/~elhadad/nlp11/prob

http://homepages.inf.ed.ac.uk/sgwater/teaching/lsa2015/lectures

https://web.stanford.edu/~jurafsky/slp3

http://www.coli.uni-saarland.de/~thorsten/tnt/

http://www.nactem.ac.uk/tsujii/GENIA/tagger/

https://nlp.stanford.edu/software/tagger.shtml

THANK YOU