Hindi Dependency Parser

Anubhav Sharma (2018114007)
Jashn Arora (2018114006)

Mentor- Pruthwik Mishra

Github Link ⟶ https://github.com/jashna14/Dependency-Parser-Hindi

Abstract

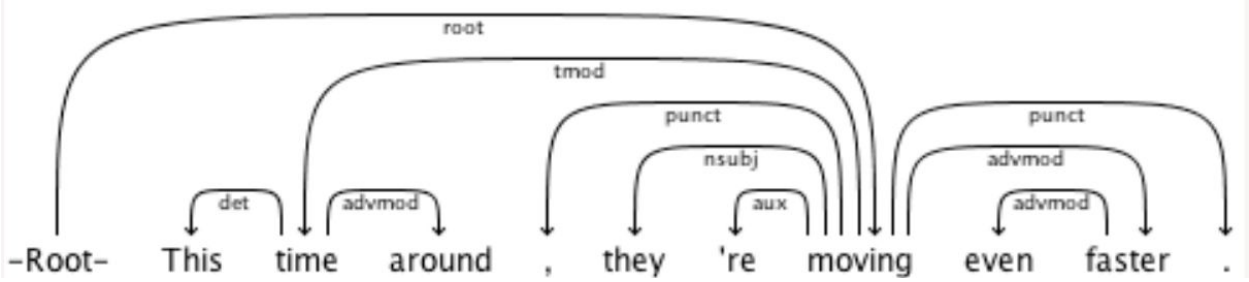
This report consists of a write up about our **approach**, **solution** and **results obtained** to the long established problem of *DEPENDENCY PARSING*. Our approaches to the problem are **multiple** and **diverse** in nature. This analysis has been done for the Hindi Language(Speakers-528 million in India).

Introduction

Before we analyse our take on the Dependency parsing , let us look at the concept of **Dependency parsing and Grammar** .

The traditional linguistic notion of grammatical relation provides the basis for the grammatical relation binary relations that comprise these dependency structures. The arguments to these head relations consist of a **head** and a **dependent**. Dependency syntax postulates that syntactic structure consists of relations between lexical items, normally binary asymmetric relations(denoted by “arrows”) called **dependencies**.

A **Dependency parser** analyzes the grammatical structure of a sentence, establishing relationships between the above *"head"* words and *words* which modify those heads(dependents).



The figure here shows the Dependency parse of a short sentence. The arrow from the word moving to the word faster indicates that faster modifies moving, and the label advmod assigned to the arrow describes the exact nature of the Dependency.

There are various approaches to Dependency parsing with each having their pros and cons , but over here we will discuss Transition Based Parsing as it's used in our implementation . Also we will discuss a BInary Classifier Algorithm called as Support Vector Machine (**SVM)** as as we trained a model for dependency parsing using this classifier.

Transition Based Parsing

It is a classic ,simple and elegant approach employing a *context-free grammar*, a *stack*, and a *list of tokens* to be parsed.

Input tokens are successively shifted onto the stack and the top two elements of the stack are matched against the right-hand side of the rules in the grammar; when a match is found the matched elements are replaced on the stack (reduced) by the non-terminal from the left-hand side of the rule being matched.

SVM

Support Vector Machines (SVM's) Support Vector Machines(SVMs) are one of the binary classifiers based on the maximum margin strategy introduced by Vapnik.

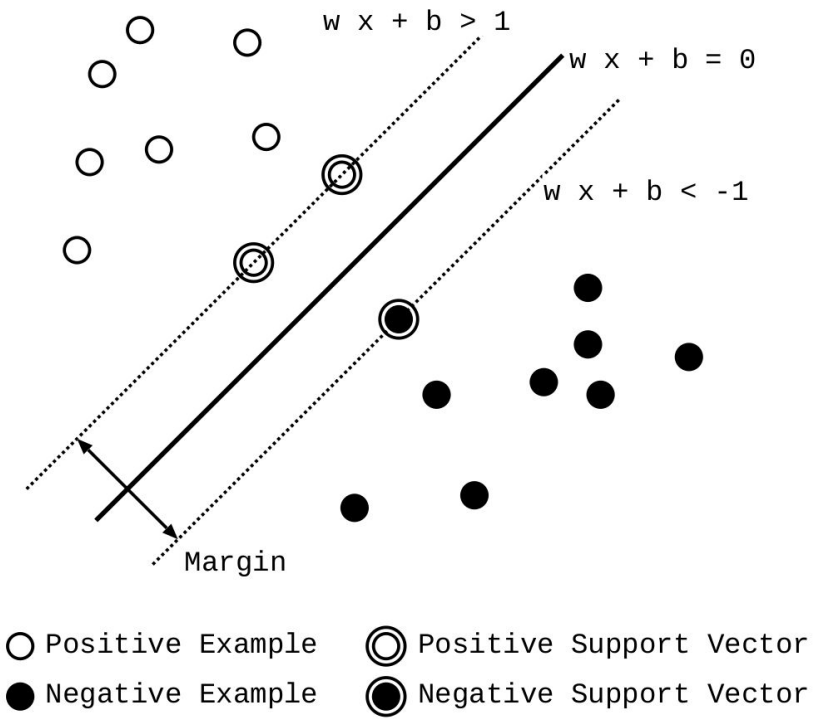


Figure Above shows an overview of how the SVM classifies the different points on the basis of the closest point to its **hyperplane**.

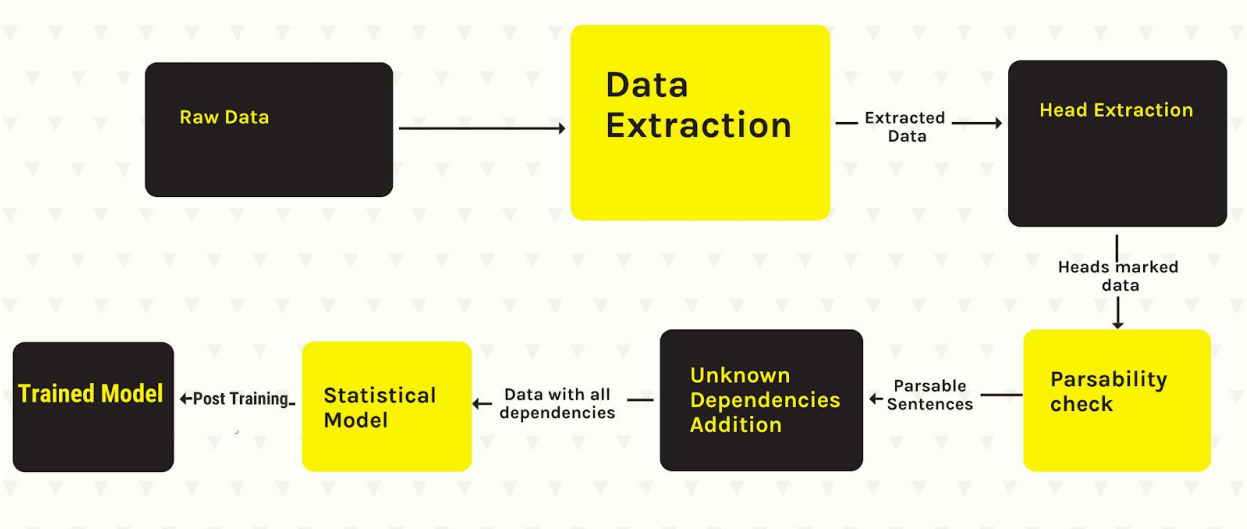
Following are the two major advantages in using SVMs for statistical dependency analysis:

1. *High generalization performance in high dimensional feature spaces*
    a. SVMs optimize the parameter w and b of the separate hyperplane based on maximum margin strategy.

b. This strategy guarantees theoretically the low generalization error for an unknown example in high dimensional feature space

2. *Learning with a combination of multiple features*
   a. It is possible by virtue of polynomial kernel functions.
   b. SVMs can deal with non-linear classification using kernel functions.
   c. SInce our analysis includes feature selection , we need a classifier which can learn from a combination of multiple features.

Short overview



Here each block represents a stage at which processing occurs.

So the basic idea of implementation is :

I. Taking data and extracting useful information
II. Computing head of chunks
III. removing non parsable sentences
IV. Develop unknown dependencies
V. Train models on the data after developing unknown dependencies

Each of these stages are explained in the further sections.

Experiment and Methodology

Data for Experiment

We have used the corpus which was taken from LTRC's dataset. The data that was provided was already a parsed data by hindi shallow parser in Shakti Standard Form(SSF). For training , testing and evaluation this parsed data was used.The data can be downloaded from http://ltrc.iiit.ac.in/treebank_H2014/

Here is an example of how the parsed sentence of Hindi (in SSF form) looks like and what all different syntactic and morphological information it provides.

<Sentence id='2'>

1 ((NP <fs name='NP' drel='k2:VGF'>

1.1 इसे PRP <fs af='यह,pn,any,sg,3,o,को,ko' posn='10' name='इसे'>

))

2 ((NP <fs name='NP2' drel='k1:VGF'>

2.1 नवाब NNC <fs af='नवाब,n,m,sg,3,d,0,0' posn='20' name='नवाब'>

2.2 शाहजेहन NNP <fs af='शाहजेहन,n,m,sg,3,o,0,0' posn='30' name='शाहजेहन'>

2.3 ने PSP <fs af='ने,psp,,,,,,' posn='40' name='ने'>

))

3 ((VGF <fs name='VGF' voicetype='active' stype='declarative'>

3.1 बनवाया VM <fs af='बनवा,v,m,sg,any,,या,yA' posn='50' name='बनवाया'>

3.2 था VAUX <fs af='था,v,m,sg,any,,था,WA' posn='60' name='था'>

))

4 ((BLK <fs name='BLK' drel='rsym:VGF'>

4.1 । SYM <fs af='।,punc,,,,,,' posn='70' name='।'>

))

</Sentence>

Explanation...

Starting of every sentence is marked with <Sentence id='x'> , where x is the id of the corresponding sentence which will be used as the sole index for locating a sentence. Also sentence ending is marked by </Sentence>.

The sentence is divided into chunks (the above sentence has 4 chunks). Each chunk has a chunk tag (marked with for the first chunk) , chunk name (marked with for the first chunk). Every chunk has a **drel** component which has two elements separated by a colon(:) (marked with for the first chunk) . The second element is the name of the chunk whose dependent is the current chunk and the first element is the dependency relationship between the two chunks.

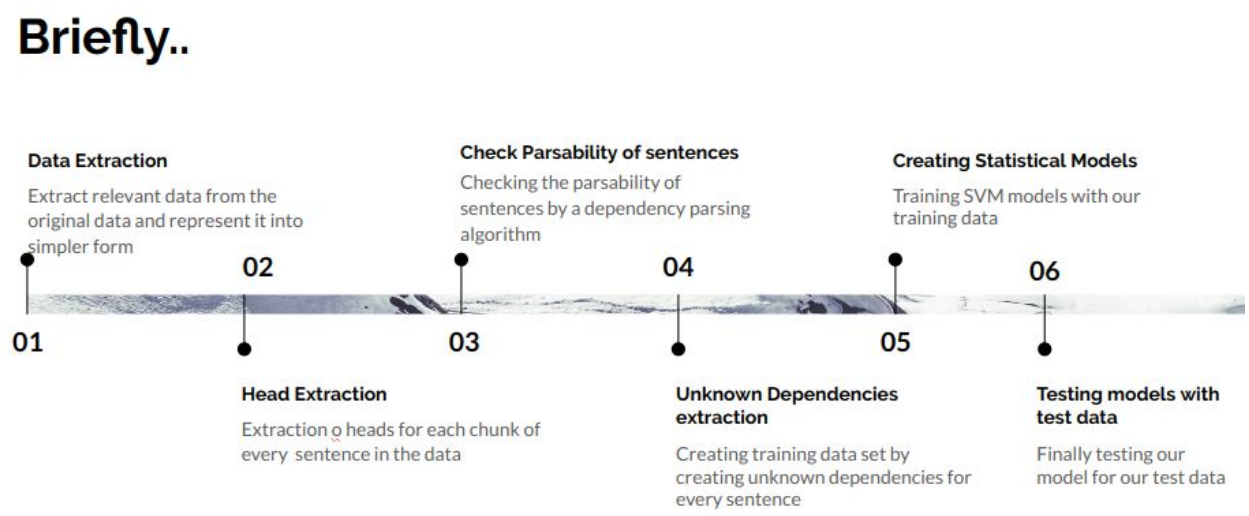
Note** : The chunk having no **drel** is the one which is the root of the whole sentence, and it is not the dependent of any chunk in the sentence.

For example, the first chunk (NP) is the dependent of the chunk with name 'VGF' (i.e third chunk).

Further every chunk has its corresponding words (marked with the highlight) , with its POS tag (marked with highlight) and its lemmatized/root form (marked with highlight). There is also the morphological information about each word in the context.

Methodology and Procedure

Our approach to solve this problem is divided into six steps. In further each subsection , all the inputs to the individual level, its processing and the outputs of that level have been elaborated.



We will go through each of the above in great detail.

Data Extraction

The process of Data Extraction simply means extracting only that data that was useful for us in the further processes and representing them in a simpler way for easy accessibility.

The format of a sentence in the SSF data (discussed above), was changed to a simpler form with only that component of the data that was useful.

Sentence Format

For example, the sentence discussed above was changed to the following format.

<Sentence id='2'>

H NP NP k2 VGF

T इसे PRP यह

H NP NP2 k1 VGF

T नवाब NNC नवाब

T शाहजेहन NNP शाहजेहन

T ने PSP ने

H VGF VGF NULL ROOT

T बनवाया VM बनवा

T था VAUX था

H BLK BLK rsym VGF

T । SYM ।

</Sentence>

Explanation...

The starting and ending of the sentence is marked by the same tags as in the SSF format. Further the lines starting with **'H'** marks the chunk start and the lines starting with **'T'** marks the words in the enclosing chunk.

The elements after 'H' are the chunk tag, chunk name , relation with the chunk whose dependent is the current chunk, and the name of the chunk whose dependent is the current chunk respectively.

The elements after 'T' are the word, its POS tag, its lemmatized/root form respectively.

Head Computation

One of the most important tasks in the initial phase of the project was to compute abd extract the head word from the words belonging to the corresponding chunk, for every chunk in the sentence.

There were in total of 9 different types chunks i.e 9 different chunk tags, these are:

- JJP
- VGF
- VGNN

- VGNF
- NEGP
- RBP
- BLK
- CCP
- NP

Observing all different types of sequences of words that constitute a tag, and applying some linguistics rules, we formed rules for finding out the head word for every chunk tag.

Rules

- **JJP**
  - First preference - ['JJ']
  - Second preference - ['QF','QC','QO']
- **VGNF**
  - Always select - ['VM']
- **VGF**
  - Always select - ['VM']
- **VGNN**
  - Always select - ['VM']
- **NEGP**
  - Always select - ['NEG']
- **RBP**
  - First preference - ['RB']
  - Second preference - ['NN','WQ']
- **BLK**
  - First preference - ['SYM']
  - Second preference - ['UNK','RP','INJ']
- **CCP**
  - First preference - ['CC']
  - Second preference - ['SYM']
- **NP**
  - First preference - ['NN','PRP','NNP']

- ➢ Second preference - ['QC','QF','QO']
- ➢ Third preference - ['NST']
- ➢ Fourth preference - ['WQ']

Procedure

For every chunk , iterate backwards in the list of words in that chunk, and the first word that has its pos tag in the list of tags which had the first preference for that chunk tag is chosen as the head.

If a head cannot be found , we follow the same procedure again, and check for pos tags of words in the the list of tags which had the second preference for that chunk tag.

Similarly go forward for third and fourth preference in case of NP chunk tag.

Data cleaning

During the process of finding the rules of extracting heads for each chunk tag, various chunks were found which did not follow the grammatical rules of chunk formation. The sentences consisting of these types of chunks were cleaned from the data set.

Types of sentences deleted were:

- All sentences with possible chunks with chunk tag **NP** and having NNC or NNPC without 'NN or NNP or PRP'.
- All sentences with possible chunks with chunk tag **NP** having JJ without 'NN or NNP or PRP or NST'
- All sentences with possible chunks with chunk tag **NEGP** having no NEG
- All sentences with possible chunks with chunk tag **FRAGP**
    - These sentences were removed as for our data set , it was not possible to form a general rule for selecting the head incase of FRAGP chunk
- All sentences with possible chunks with chunk tag **VGF** having no VM
- All sentences with missing word i.e having pos as **NULL__x** where 'x' can be any one of the chunk tags.

Sentence Format

After finding out the head word for every chunk the sentence format was as follows,

<Sentence id='2'>

इसे नवाब शाहजेहन ने बनवाया था ।

यह नवाब शाहजेहन ने बनवा था ।

PRP NNC NNP PSP VM VAUX SYM

इसे शाहजेहन बनवाया ।

यह शाहजेहन बनवा ।

NP NP VGF BLK

PRP NNP VM SYM

NP NP2 VGF BLK

H इसे यह NP PRP NP k2 VGF

H शाहजेहन शाहजेहन NP NNP NP2 k1 VGF

H बनवाया बनवा VGF VM VGF NULL ROOT

H । । BLK SYM BLK rsym VGF

</Sentence>

The starting and ending of the sentence was marked with the same tags as earlier.

The first line after the starting tag is the sentence. The second line is the sentence with its words in the lemmatized/root form. The third line is the sequence of pos tags for every word of the sentence.

The fourth line is the sequence of head words for every chunk in the sentence. The fifth line is the sequence of head words in their lemmatized/root form. The sixth line is the sequence of corresponding chunk tags. The seventh line is the sequence of tags of the words selected as the heads for every chunk.

The eighth line is the sequence of the names of the chunk.

Now for the sentences starting with the identifier ‘H’ marking each chunk in the sentence, the format is as follows:

- The first word after ‘H’ is the word chosen as the head of the chunk, the next word is its lemmatized/root form of the word followed by the chunk tag, then the pos tag of the word chosen as the head.
- The last three elements are the same as it were in the previous format of the sentences.

Parsability check

This task was carried out to check the parsability of the sentences. As one doesn’t want his model to train on the wrong dataset, we wanted to remove all those sentences which were non parsable as in the sentences which were parsed wrong by the shallow hindi parser.

Also, these non parsable sentences can be used in further research to improve the hindi parser.

There are many parsing algorithms available. We chose Arc-Eager for checking the parsability of the sentences.

Arc Eager Algorithm

The Arc-Eager system for transition-based dependency parsing is widely used in natural language processing. The approach it follows is of greedy deterministic dependency parsing.

Here we will discuss the algorithm to deterministically parse the sentence and use it to check the parseability of the sentences.

A parser configuration consists of a stack σ, a buffer β, and a set of arcs A. The initial configuration for parsing a sentence x = w1 , . . ., wn has an empty stack, a buffer containing the words w1 , . . . , wn, and an empty arc set.

Initialization - Now with the sentence coming as input gets stored in the buffer and the stack is initialised with a “ROOT” node .

Progression - Now before we move forward we define the four steps that are to follow during the progression stage:

Steps Available

1. **Left Arc-** If *i* and *j* are the top of stack and buffer respectively and if the left arc operation is selected , the stack top is popped and a relationship is appended in the set A with the tag ⇒ (*i*, *j*,R)
2. **Right Arc-** If *i* and *j* are the top of stack and buffer respectively and if the right arc operation is selected , then the buffer element is popped and pushed on the stack and a relationship is appended in the set A with the tag ⇒ (*i*, *j*,L)
3. **Shift -** If *i* is the top of the buffer and if the shift operation is selected , then then the buffer top is popped and pushed on the stack.
4. **Reduce-** If *i* is the top of the Stack and if the reduce operation is selected ,then the stack top is popped.

Over here **L** and **R** denote the relative position of head i.e **L** denotes that head is at left and **R** denotes that head is at right .

Pre Conditions for each Step

1. **Left arc -** If *i* and *j* are the top of stack and buffer respectively , Before appending In the set A of arcs , check if the child (stack top) has any other head in the set A, i.e if any arc (left , right) terminates on *i* or not . If not ⇒ perform **left arc** operation , else ⇒ look for other options.
2. **Right arc-** If *i* and *j* are the top of stack and buffer respectively , Before appending In the set A of arcs , check if the child (stack top) has any other head in the set A, i.e if any arc (left , right) terminates on *j* or not . if not ⇒ perform **right arc** operation , else ⇒ look for other options.
3. **Reduce -** if If *i* and *j* are the top of stack and buffer respectively then we check for these 2 conditions:
    a. Firstly, We check if there is any link of any element of *k* with j, in the set A. IF we find one , then we check if *k* appeared before *i* in sentence

or not . If *k* didn't appear early then we come to the conclusion that **reduce** cannot be taken . But if for all such *k*'s *k* is less than *i* , we perform **reduce** operation.

b. If condition a is satisfied , then we check that for **reduce** operation to occur , there must be at least one dependency pair in the set A , with *i* as the head , i.e (*i*,*x*,L) or (*x*,*i*,R) must be an element of set A (where *x* can be any token from the given sentence).

4. **Shift -** If buffer is empty ⇒ do not perform **shift** . if it is empty ⇒ perform **shift** operation.

Termination - If both stack and buffer are empty , we terminate.

NOTE- we first look to perform **left arc operation ,then we look to perform **right arc** operation and then we look to perform anything among **shift** and **reduce** , whose conditions are satisfied.**

At any step if we find out that during the progression stage , if at any point we cannot find the way to take any step between SHIFT , REDUCE , LEFT ARC , RIGHT ARC then we declare the sentence as non parsable.

Before applying arc eager to the sentences, the format of the sentences was changed to make it easier to apply arc eager.

Sentence Format

<Sentence id='2'>

इसे नवाब शाहजेहन ने बनवाया था ।

यह नवाब शाहजेहन ने बनवा था ।

PRP NNC NNP PSP VM VAUX SYM

इसे शाहजेहन बनवाया ।

यह शाहजेहन बनवा ।

NP NP VGF BLK

PRP NNP VM SYM

NP NP2 VGF BLK

H इसे यह NP PRP NP k2 VGF ; H बनवाया बनवा VGF VM VGF NULL ROOT ; R ; k2

H शाहजेहन शाहजेहन NP NNP NP2 k1 VGF ; H बनवाया बनवा VGF VM VGF NULL ROOT ; R ; k1

ROOT ; H बनवाया बनवा VGF VM VGF NULL ROOT ; L ; ROOT

H बनवाया बनवा VGF VM VGF NULL ROOT ; H । । BLK SYM BLK rsym VGF ; L ; rsym

</Sentence>

The format almost remains the same, but the lines starting with 'H' which contained information about that chunk, now also contains information about the chunk it is related to , followed by the marking of the head of that relationship among the two chunks.

L means that the left one is the head, **R** means that the right one is the head. This was followed by the dependency relationship between the two chunks as k1/k2 ... . All this information is stored for every chunk separated by a semicolon ';'.

Note** : The chunk on the left always comes first in the sentence than the chunk on the right in every line starting with 'H'.

The word that is the head of the sentence is shown with the relationship of **'ROOT'** with a dummy variable **ROOT** that is inserted in the stack in arc eager as we know that the head word of the sentence is the dependent of that dummy variable in arc eager.

The chunks having 'L' relationship will lead to the formation of **Right arc** in arc eager. Similarly, The chunks having 'R' relationship will lead to the formation of **Left arc** in arc eager.

Unknown Dependencies Extraction

For training our model, we wanted the list of all types of dependencies in our sentences.

We had the list of all known dependencies i.e the pair of words in the sentence that are related to each other and also the head word among them was marked by the **L** or **R.**

We also need to make some pairs of words that are not related to each other in the sentence to train our model uniformly so that it can also predict the pair of words which are not related.

These unknown dependencies were created by taking few of the unrelated pairs from the sentence by our algorithm. The head position was marked by **'U'** and their dependency relation was marked as **'NULL'.**

The sentence format after inserting the unknown dependencies is shown below.

Sentence Format

H इसे यह NP PRP NP k2 VGF ; H बनवाया बनवा VGF VM VGF NULL ROOT ; R ; k2

H शाहजेहन शाहजेहन NP NNP NP2 k1 VGF ; H बनवाया बनवा VGF VM VGF NULL ROOT ; R ; k1

ROOT ; H बनवाया बनवा VGF VM VGF NULL ROOT ; L ; ROOT

H बनवाया बनवा VGF VM VGF NULL ROOT ; H । । BLK SYM BLK rsym VGF ; L ; rsym

H इसे यह NP PRP NP k2 VGF ; H शाहजेहन शाहजेहन NP NNP NP2 k1 VGF ; U ; NULL

H शाहजेहन शाहजेहन NP NNP NP2 k1 VGF ; H । । BLK SYM BLK rsym VGF ; U ; NULL

H इसे यह NP PRP NP k2 VGF ; H । । BLK SYM BLK rsym VGF ; U ; NULL

ROOT ; H शाहजेहन शाहजेहन NP NNP NP2 k1 VGF ; U ; NULL

ROOT ; H इसे यह NP PRP NP k2 VGF ; U ; NULL

Statistical Models Creation

Given that now the data consists of all the known and unknown dependencies and is in the form where a model can be trained over it , we train a binary classifier in the form of **SVM** (reasons for selecting SVM have been mentioned in the Introduction section) . We will train multiple models each differentiating with others on the basis of their feature vectors.

The procedure of parsing a sentence goes, in two steps:

Step 1: Predicting which all pairs of words in the sentence are related/unrelated and which one is the head of the relationship in case the words are related. (Labels L,R,U were used to mark these relations)

Step 2: For the pairs of words which are related, predicting the dependency relationship between the two. (Labels k1, k2 etc. were used to mark these relationships).

So, we made five models to predict the labels L,R,U and six models for predicting the dependency relations marked with label k1, k2 etc. .

Feature Representation

Following are the representation of features for a chunk.

Features	Meaning
$word_i$	Head word of the chunk i
pos_i	POS of head word of chunk i
$chunk\_tag_i$	Tag of the chunk i
psp_i	The postposition in the list of words in chunk i
L/R	The label for marking the head among the two related chunks

Note **: For psp , the first step of extracting data was done again to extract psp for every chunk for only those sentence that were found parsable by the arc eager in step 3

Label representation

Label	Meaning
L/R/U	Marking the head word among the two chunks
k1/k2..	Marking the dependency relationship between the chunks

Models

Model Number	Feature vector	Label
1	$word_i + pos_i + chunk\_tag_i + word_j + pos_j + chunk\_tag_j$	L/R/U
2	$word_i + pos_i + chunk\_tag_i + word_j + pos_j + chunk\_tag_j$	k1/k2..
3	$word_i + pos_i + word_j + pos_j$	L/R/U
4	$word_i + pos_i + word_j + pos_j$	k1/k2..
5	$word_i + word_j$	L/R/U

6	$word_i + word_j$	k1/k2..
7	$word_i + pos_i + psp_i + word_j + pos_j + psp_j$	L/R/U
8	$word_i + pos_i + psp_i + word_j + pos_j + psp_j$	k1/k2..
9	$word_i + pos_i + chunk\_tag_i + psp_i + word_j + pos_j + chunk\_tag_j + psp_j$	L/R/U
10	$word_i + pos_i + chunk\_tag_i + psp_i + word_j + pos_j + chunk\_tag_j + psp_j$	k1/k2..
11	$word_i + pos_i + chunk\_tag_i + psp_i + L/R + word_j + pos_j + chunk\_tag_j + psp_j + L/R$	k1/k2..

Testing models

The testing of all the models created was done on the test set. The metric analysis of is discussed below.

Evaluation Metrics

Model evaluation metrics are required to **quantify model performance**. The choice of evaluation metrics depends on a given machine learning task. So For the performance Analysis of our Model , we used certain standard evaluation methods which are **fit** for **classification metric** analysis (Given that our model is a binary classifier).

Following are the metrics that we calculated :

1. ***Precision -*** The precision is the ratio tp / (tp + fp) where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.
2. ***Recall -*** The recall is the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples.
3. ***F1_score-*** The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0. The

relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

F1 = 2 * (precision * recall) / (precision + recall)

4. ***Confusion Matrix-*** By definition a confusion matrix is such that is equal to the number of observations known to be in group and predicted to be in group .Thus in binary classification, the count of true negatives is , false negatives is , true positives is and false positives is .

Now there are three flags which compute our data ,and are explained in the following sentences :

1. **Weighted**- Calculates metrics for each label, and finds their average weighted by support (the number of true instances for each label). This alters ‘macro’ to account for label imbalance; it can result in an F-score that is not between precision and recall.
2. **Macro**- Calculates metrics for each label, and finds their unweighted mean. This does not take label imbalance into account.
3. **Micro**-Calculated by counting the total true positives, false negatives and false positives.

Following are their values found on the testing set for **first phase of testing** for the above mentioned metrics:

HERE THE ‘-’ MEANS THAT THE VALUE OBTAINED FOR THAT PARTICULAR PARAMETER HAD NO VALUES REGARDING A PARTICULAR FEATURE , SO FLAGS IN WHICH COMPUTATION OF METRIC HAPPENS BY SEGREGATION , THEY WOULD THROW A WARNING , THUS THEIR VALUES ARE NOT CONSIDERED.

Values For Macro :

Model Number	Precision	Recall	F1_Score
1	0.754	0.7829	0.7657
2	-	-	0.3787
3	0.75296	0.76241	0.75518
4	-	-	0.37467
5	0.74849	0.75267	0.74876
6	-	-	0.34668
7	0.75516	0.76373	0.75691
8	-	-	0.444197
9	0.75533	0.78492	0.76730
10	-	-	0.45477
11	-	-	0.471184

Values for Micro:

Model Number	Precision	Recall	F1_Score
1	0.7766	0.7766	0.7766
2	0.7715	0.7715	0.7715
3	0.77301	0.77301	0.77301
4	0.76829	0.76829	0.76829
5	0.767155	0.767155	0.767155
6	0.74517	0.74517	0.74517
7	0.77478	0.77478	0.77478
8	0.85022	0.85022	0.85022

9	0.778431	0.778431	0.778431
10	0.853166	0.853166	0.853166
11	0.86321	0.86321	0.86321

Values for Weighted

Model Number	Precision	Recall	F1_Score
1	0.78759	0.7766	0.77916
2	-	-	0.761798
3	0.78274	0.7730	0.77523
4	-	-	0.75849
5	0.774723	0.7671	0.76913
6	-	-	0.73413
7	0.78445	0.77478	0.77697
8	-	-	0.84613
9	0.78949	0.77843	0.78095
10	-	-	0.84922
11	-	-	0.86033

Findings of metric analysis

After looking at the results , we drew some conclusions that were extracted on the basis of our models' performance :

1. The values for micro flag are the same as the number of False positives and false negatives are same , thus ensuring same precision and recall , which inturn ensures the same F1_score.

2. The values of F1_score when computed for macro flag are really poor , when we compute them to find the dependencies (k1,k2 etc .) . The probable reason being that the testing data contains a lot of labels for one feature as compared to the other. So the one with lesser counts further pushes down the overall value. Whereas when weights are given to each input label (feature) , then the results improved.
3. Our values for f1_score and other parameters while computing dependencies came better when the psp was considered , as in Hindi , postposition acts as the marker for **cases** . For example: In Hindi " ने" always detects the presence of karta case relationships.

k1	karta (doer/agent/subject)
k2	karma (object/patient)
k3	karana (instrument)
k4	sampradaana (recipient)
k5	apaadaana (source)
k7t	kaalaadhikarana (location in time)
k7p	deshadhikarana (location in space)
k7	vishayaadhikarana (location elsewhere)
ras	upapada__ sahakaarakatwa (associative)
rd	prati upapada (direction)
rh	hetu (cause-effect)
k*u	saadrishya (similarity)
rt	taadarthya (purpose)
k1s	vidheya karta (karta samanadhikarana)
k2s	vidheya karma (karma samanadhikarana)
r6	shashthi (possessive)
pk1	prayojaka karta (causer)
mk1	madhyastha karta (causer2)
jk1	prayojya karta (causee)
nmod	Noun modifiers
jjmod	Adjectival modifiers
adv	kriyaavisheshana ('manner adverbs' only)
rad	Address words
ccof	Conjunct of relation
pof	Part of relation
nmod_relc	Noun modifier of the type relative clause
jjmod_relc	Adjectival modifier of the type relative clause.
rbmod_relc	Adverbial modifier of the type relative clause

This image describes the relationship between the karaks and dependencies , thus giving us the validation for good results.

4. Our values for f1_score and other parameters while computing direction of dependencies(L,R,U) did not improve when the psp was considered. The reason being Hindi is a free order language and the relative direction can change. For example :

   राम ने श्याम को मारा ।

   शयाम ने राम से मार खायी ।

   So in these sentences the presence of "ने" cannot help us in finding the directions of

   k1 relationship.

In further work,we tried to experiment and find out how many sentences after all the analysis, we can get correct and complete dependencies from the testing set.

Future Work

- The model could be trained using better features which would contain the information of the words in the context of the chunks. These would definitely improve the accuracies of the models as context words may also define the relationship between chunks.
- The models could be trained keeping in mind the sentence boundary which would bring in the rules of dependency parsing that must be followed by a parsed sentence into action.

References

[1] http://www.phontron.com/slides/nlp-programming-en-11-depend.pdf

[2] https://arxiv.org/pdf/1901.10457.pdf

[3] Hiroyasu Yamada, Yuji Matsumoto, 2003."STATISTICAL DEPENDENCY ANALYSIS WITH SUPPORT VECTOR MACHINES",Graduate School of Information Science

[4] https://cl.lingfil.uu.se/~sara/kurser/5LN455-2014/lectures/5LN455-F8.pdf

[5] Malt Parser - http://lrec-conf.org/proceedings/lrec2006/pdf/162_pdf.pdf

[6] https://www.mitpressjournals.org/doi/pdf/10.1162/coli.07-056-R1-07-027

[7] https://dl.acm.org/doi/10.5555/2002736.2002777

[8] https://www.aclweb.org/anthology/J14-2002.pdf

[9] https://www.aclweb.org/anthology/L16-1727.pdf

[8] http://universaldependencies.org/conll17/evaluation.html

[9] https://www.aclweb.org/anthology/W12-5617.pdf

[10]http://cdn.iiit.ac.in/cdn/ltrc.iiit.ac.in/icon/2009/nlptools/CR/all-papers-toolscontest.pdf#page=32

Acknowledgement Note

We really want to thank Pruthwik sir for his fruitful guidance . He always helped us whenever we had difficulty in figuring out the next step.

Also we would like to thank Manish sir for providing us with such a wonderful opportunity to work on this brilliant problem . We really learned a lot in the process and thank you sir.

We would also like to thank Alok for his critical analysis regarding the project and his “Always ready to help nature “ . You are really one of the finest TA . Thank You