Project Report
Dependency Parsing for Hindi Language

1 Problem Statement

Dependency parsing is the process of analyzing the grammatical structure of a sentence based on the dependencies between the words in a sentence. In Dependency parsing, various tags represent the relationship between two words in a sentence. We have to predict the transitions to construct the dependency tree.

2 Project Progress

2.1 Baseline-(30^{th} March, 2021)

Upto this point in our project , we have implemented a baseline with a small part of hopefully our baseline+ dependency parser. The task is to perform transition based dependency parsing for the Hindi Language. Transition based parsers use features extracted from the given dataset to predict the next action to be taken while constructing the dependency tree. To get this task done, the project follows a particular flow of task which are explained below:

2.1.1 Literature Search

Since it is a vast topic, a lot of literature search had to be done. From understanding the bigger picture to getting the dataset and working on it, a lot had to be done. We read a few papers who have done this in the past, but mostly we focused on papers which gave a good idea of extracting features from the data. We went with the paper by Yue Zhang and Joakim Nivre titled "`Transition-based Dependency Parsing with`

from single words
S_0wp; S_0w; S_0p; N_0wp; N_0w; N_0p; N_1wp; N_1w; N_1p; N_2wp; N_2w; N_2p;
from word pairs
S_0wpN_0wp; S_0wpN_0w; S_0wN_0wp; S_0wpN_0p; S_0pN_0wp; S_0wN_0w; S_0pN_0p N_0pN_1p
from three words
$N_0pN_1pN_2p$; $S_0pN_0pN_1p$; $S_{0h}pS_0pN_0p$; $S_0pS_{0l}pN_0p$; $S_0pS_{0r}pN_0p$; $S_0pN_0pN_{0l}p$

Table 1: Baseline feature templates.
w – word; p – POS-tag.

Figure 1: A sample feature set discussed by Zhang and Nivre in their paper.

`Rich Non-local Features`"[1].

They gave us a good idea of the task workflow and which features to use for our baseline model.

The other sources we referred was the Stanford NLP book, which gave us a good idea about the algorithmic structure and the required lingo to understand different papers and articles throughout the internet.

We understood what TreeBanks are and the various types of dependency relations and how to gather the dataset for our task.

An in-depth understanding with working example was shown by Dr. Pawan Goyal in Week-6 of his NLP lecture series on NPTEL[2]. This was an excellent resource of studying this topic and formed the basis of our code implementation. This set of lectures helped us in understanding the proper workflow and what oracles are and how to build them. Understood about projectivity constraints and the need to check them before proceeding further and how ML classifiers work and what needs to be fed to them for getting the task done.

2.1.2 Exploration of the task and Dataset

We downloaded the Dataset from the `Universal Dependencies Treebank for Hindi Language`. It contained the Hindi treebanks in the CoNLL-U format.

```
# sent_id = train-s3
# text = इसका प्रवेश द्वार दो मंजिला है ।
1   इसका  यह   PRON   PRP Case=Acc,Gen|Gender=Masc|Number=Sing|Person=3|Poss=Yes|PronType=Prs 3   nmod    _   Vib=का Tam=kA|ChunkId=NP|ChunkType=head|Translit=isakā
2   प्रवेश  प्रवेश  NOUN   NNC Case=Nom|Gender=Masc|Number=Sing|Person=3   3   compound    _   Vib=0|Tam=0|ChunkId=NP2|ChunkType=child|Translit=praveśa
3   द्वार  द्वार  NOUN   NN  Case=Nom|Gender=Masc|Number=Sing|Person=3   5   nsubj   _   Vib=0|Tam=0|ChunkId=NP2|ChunkType=head|Translit=dvāra
4   दो  दो  NUM QC  NumType=Card    5   nummod  _   ChunkId=JJP|ChunkType=child|Translit=do
5   मंजिल  मंजिल  ADJ JJ  _   0   root    _   ChunkId=JJP|ChunkType=head|Translit=ma⍰jilā
6   है  है  AUX VM  Mood=Ind|Number=Sing|Person=3|Tense=Pres|VerbForm=Fin|Voice=Act 5   cop _   Vib=है Tam=hE|ChunkId=VGF|ChunkType=head|Stype=declarative|Translit=hai
7   ।  ।  PUNCT  SYM _   5   punct   _   ChunkId=BLK|ChunkType=head|Translit=.
```

Figure 2: Caption

The detailed information about all the fields in the CoNLL-U format can be found here.

To understand what dependency trees look like we explored another feature provided by the Universal Dependencies group, which is its CoNLL-U viewer. This visualization aid helped in understanding the treebank dataset and how the fields are connected to each other to make the whole structure look like a tree with their corresponding dependency relations.

For example, for one of the sentences in the dataset, the dependency tree looks as the one shown below.

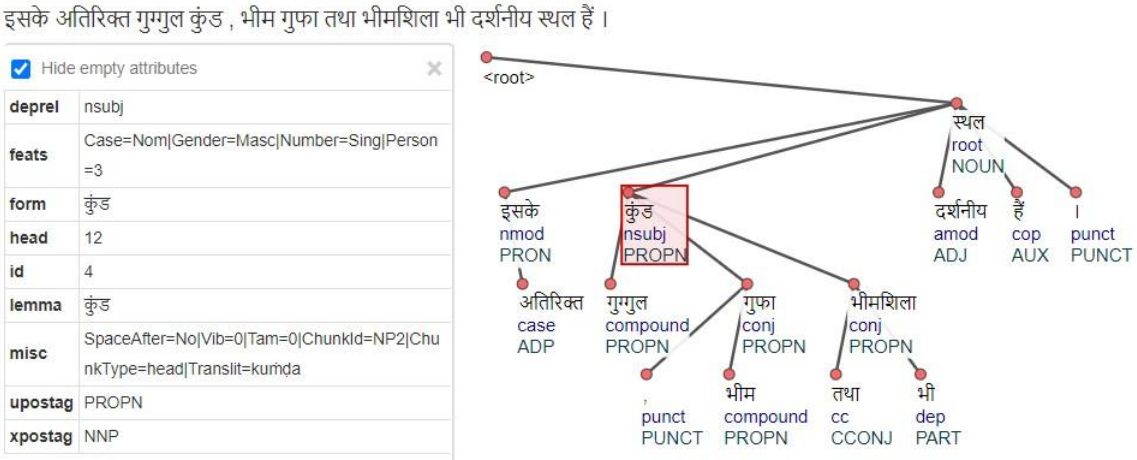

Figure 3: Sample Dependency Tree in CoNLL-U Viewer.

In the above example the complete structure of the dependency tree is shown, and for the word "कुंड" we can see the corresponding feature fields in Figure 3. This helped us in appreciating the individual fields of the datasets and helped us a lot in extracting the features.

2.1.3 Implementation

We learned from the sources mentioned in 2.1.1, and proceeded with the implementation of the various parts of the task. The various parts of the parser and the workflow which we followed are explained below:

- **Part 1 : Parsing and extracting required fields from the data** The CoNLL-U dataset format has 10 fields for each token of the sentence.

  We parsed the file by extracting only these token lines, and selected the fields 1,2,5,7,8 which correspond to the sentenceID, Word form of the token, Language specific POS tag, the head of the current word, and the dependency relation of the word with its head.

  Code sample for extracting the different line tokens is shown below :

```
def extract_line_tokens(self,line):
    line_id,word,stem_word,u_pos,x_pos = line[0],line[1],line[2],line[3],line[4]
    feat,head,dep_rel,dep,misc = line[5],line[6],line[7],line[8],line[9] return
    int(line_id),word,x_pos,int(head),dep_rel
```

- **Part 2: Building the Oracle and creating training data.** We followed the following architecture stated in the Stanford NLP book to achieve the task. A

function DEPENDENCYPARSE(*words*) **returns** dependency tree

state ← {[root], [*words*], [] } ; initial configuration
while *state* **not final**
  t ← ORACLE(*state*) ; choose a transition operator to apply
  state ← APPLY(*t*, *state*) ; apply it, creating a new state
return *state*

Figure 4: Algorithm Architecture for Dependency Parser Code

sample that invokes the above algorithm is shown below:

```
dp = DependencyParser(train_treebank)
x_train, y_train = dp.build_oracles()
```

The build oracle function takes input the treebank and checks the tree is projective or not, the proceeds ahead for extracting the configuration states and gold transitions for the current tokens. After extracting the all the possible configuration states accompanied by their corresponding gold standard transitions, features and labels are extracted for each configuration state. The labels remain the same as the gold transitions and features are extracted for each corresponding configuration state.

The gold standard transitions are extracted based on the following conditions :

LEFTARC(r): **if** $(S_1\ r\ S_2) \in R_p$

RIGHTARC(r): **if** $(S_2\ r\ S_1) \in R_p$ **and** $\forall r', w\ s.t. (S_1\ r'\ w) \in R_p$ **then** $(S_1\ r'\ w) \in R_c$

SHIFT: **otherwise**

Figure 5: Conditions for getting the gold standard transitions based on configuration states(Stack, Arc and Buffer)

Implementation of the above algorithm to check for transition states, the function returns boolean values True or False for the corresponding conditions.

```
def check_condition(self,buffer,stack,dependencies = {},condition=None):

    if(condition == '0'): return len(buffer) is 0 and len(stack) is
        1 and stack[0] == 0

    elif(condition      ==      '1'):
        #condition for shift
        return len(stack) < 2 and len(buffer) > 0 #stack has only root and items
                                                    are still in buffer

    elif(condition == '2'): #condition for left
        arc stack_top = stack[-1] stack_2nd =
        stack[-2]

       return dependencies.get(stack_top) is not None and (stack_top,stack_2nd) in
                                                        dependencies[stack_top]

    elif(condition == '3'): #condition for right
        arc stack_top = stack[-1] stack_2nd =
        stack[-2]

       return dependencies.get(stack_2nd) is not None and (stack_2nd,stack_top) in
                                                      dependencies[stack_2nd]
```

Implementation of baseline feature extraction function which we fed to our classifier is shown below :

```
    def get_features_labels(self,current_configuration,current_transtion,tokens_dict,
                                                        POS_tags):
        def get_stack_features(idx,stack,tokens_dict,POS_tags):
            if idx == 'top':
                return tokens_dict[stack[-1]],POS_tags[stack[-1]]
            elif idx == '2nd_top':
                return tokens_dict[stack[-2]],POS_tags[stack[-2]]

        def get_buffer_features(buffer,tokens_dict,POS_tags): return
            tokens_dict[buffer[-1]],POS_tags[buffer[-1]]

        def get_two_word_features(tokens_dict,stack,POS_tags): return
            tokens_dict[stack[-1]],POS_tags[stack[-1]],POS_tags[stack[-2]]

        def insert_to_features(features,feat_i,feat_j,feat_k):
            features[feat_i] = 1 features[feat_j] = 1
            features[feat_k] = 1

        label = current_transtion buffer,stack,arcs =
        current_configuration features = defaultdict()

        buffer_len = f'buffer_length_{len(buffer)}' stack_len =
        f'stack_length_{len(stack)}'

        if len(stack) > 0:
            top_token,top_pos = get_stack_features('top',stack,tokens_dict,POS_tags)

            top_feature_1 = f'stack_top_{top_token}' top_feature_2 =
            f'stack_pos_{top_pos}'
            top_feature_3 = f'stack_top_pos_{top_token}_{top_pos}'

            insert_to_features(features,top_feature_1,top_feature_2,top_feature_3)

        if len(stack) > 1:
            top2nd_token,top2nd_pos = get_stack_features('2nd_top',stack,tokens_dict,
                                                        POS_tags)

            top2nd_feature_4 = f'stack2nd_top_{top2nd_token}' top2nd_feature_5 =
            f'stack2nd_pos_{top2nd_pos}'
            top2nd_feature_6 = f'stack2nd_top_pos_{top2nd_token}_{top2nd_pos}'

            insert_to_features(features,top2nd_feature_4,top2nd_feature_5, top2nd_feature_6)

        if len(buffer) > 0:
            buf_token,buf_pos = get_buffer_features(buffer,tokens_dict,POS_tags)

            buf_feature_7 = f'buffer_top_{buf_token}' buf_feature_8 =
            f'buffer_pos_{buf_pos}'
            buf_feature_9 = f'buffer_top_pos_{buf_token}_{buf_pos}'
            insert_to_features(features,buf_feature_7,buf_feature_8,buf_feature_9)
```

```
    if len(stack) > 1 :
        stack_top , pos_top , pos_2nd_top = get_two_word_features(tokens_dict,stack
                                                    ,POS_tags)

                    two_word_feature_10 = f'stack_top_pos_top_pos_2ndtop_{stack_top}_{pos_top}_
                                                    {pos_2nd_top}'
        features[two_word_feature_10]     =     1     return
    features,label
```

We used 10 features for each configuration state as inputs for our baseline model, which are:

1. Stack top.
2. POS tag of Stack top.
3. Joint occurrence of current stack top with its POS tag.
4. Stack 2^{nd} top.
5. POS tag of Stack 2^{nd} top.
6. Joint occurrence of current stack 2^{nd} top with its POS tag.
7. Buffer top.
8. POS tag of Buffer top.
9. Joint occurrence of current buffer top and its POS tag.
10. Joint occurrence current Stack top and its POS tag, and POS tag of stack 2^{nd} top.

The 10^{th} feature is a two-word feature and 1^{st} to 9^{th} are one word features.

The complete function to build oracles and return the training data is shown below :

```
    def build_oracles(self,):
        current_tokens = []
        tokens_dict = {} POS_tags
        = {}

        tokens_dict[0]= 'root' POS_tags[0]=
        'root'

        for current_line in self.train_treebank:
            if len(current_line) < 2 : # useful fields if len(current_tokens) > 0 :
                #have        found        some        tokens        before        if
                self.check_projectivity(current_tokens) == True: #if current tree is
                projective
                        cs,gt = self.extract_configs_transitions(current_tokens)
```

```
                    for i in range(len(cs)):
                        print(f'{cs[i]}------{gt[i]}')
                features , labels = self.get_features_labels(cs[i],gt[i], tokens_dict,




                        self.features.append(features) self.labels.append(labels)


                current_tokens = []
                tokens_dict = {} POS_tags =
                {}

                tokens_dict[0]= 'root'
                POS_tags[0]= 'root' continue

        if current_line[0].startswith('#'):
            continue

        tok_id,word,x_pos,head_id,dep_rel = self.extract_line_tokens(current_line)
        current_tokens.append((tok_id,word,x_pos,head_id,dep_rel))
        tokens_dict[tok_id] = word POS_tags[tok_id] = x_pos

    X_train,y_train = self.convert_to_training_data()

    print('finished')
    return X_train,y_train,self.features
```

- **Part 3 : Training and choice of classifiers**

  After extracting the features and labels, the next step was to train a classifier for classification.

  On examining the labels, we found that there were total 53 unique labels, which included the shift, left_arc transitions, and right_arc transitions.

  Since there are more than 2 classes, it is a multiclass classification problem. We need to choose classifiers which can return probability scores of all the classes, because it will help us in evaluating the classifier on the test data later.

  So, to implement this we experimented with different classifiers namely Logistic Regression with different penalties, and Bernoulli Naïve Bayes classifier.

Figure 6: The list of all the unique labels(transitions) as returned by the oracle.

Figure 7: A sample set of features which are used as input for the classifier

Since the feature matrix was very large in size, it could not fit in our usual RAM's, so we converted it into sparse matrix format, and chose the classifiers which can take sparse matrix as an input and return probability scores for evaluation in the output.

2.2 Baseline+ Model (20th April, 2021)

Apart from the feature extraction part and little changes in the evaluation section, most of the things remain same.

The feature extraction was done in accordance with the paper Transition-based Dependency Parsing with Rich Non-local Features by Zhang and Nivre [1]. The features discussed in this paper are not directly available but instead are extracted from the dataset and are calculated on the fly while parsing. The features capture the context in a much better when included with the features extracted for the baseline model.

distance
S_0wd; S_0pd; N_0wd; N_0pd; S_0wN_0wd; S_0pN_0pd;
valency
S_0wv_r; S_0pv_r; S_0wv_l; S_0pv_l; N_0wv_l; N_0pv_l;
unigrams
$S_{0h}w$; $S_{0h}p$; S_0l; $S_{0l}w$; $S_{0l}p$; $S_{0l}l$; $S_{0r}w$; $S_{0r}p$; $S_{0r}l$;$N_{0l}w$; $N_{0l}p$; $N_{0l}l$;
third-order
$S_{0h2}w$; $S_{0h2}p$; $S_{0h}l$; $S_{0l2}w$; $S_{0l2}p$; $S_{0l2}l$; $S_{0r2}w$; $S_{0r2}p$; $S_{0r2}l$; $N_{0l2}w$; $N_{0l2}p$; $N_{0l2}l$; $S_0pS_{0l}pS_{0l2}p$; $S_0pS_{0r}pS_{0r2}p$; $S_0pS_{0h}pS_{0h2}p$; $N_0pN_{0l}pN_{0l2}p$;
label set
S_0ws_r; S_0ps_r; S_0ws_l; S_0ps_l; N_0ws_l; N_0ps_l;

Table 2: New feature templates.
w – word; p – POS-tag; v_l, v_r – valency; l – dependency label, s_l, s_r – labelset.

Figure 8: Feature set for the baseline+ model by Zhang and Nivre.

The additional features we tried implementing using the paper are:

1. Distance between stack top and current buffer head: It has been already used in maximum spanning tree parsing methods, it is indirectly included in the left-arc and right-arc actions, but we include this distance of stack top and buffer head along with their word and POS tags information separately in the feature set.
2. Valency of stack top and buffer head: We calculate the number of left and right modifiers separately, calling them left valency and right valency, respectively. They are combined with the word and POS-tag of stack top and buffer head to form new feature templates.
3. Some third order / trigram mixed features: Higher-order context features have been used by graph-based dependency parsers to improve accuracies.

4. Label feature: Labels named with their transition type and dependencies, instead of just transition type.

The implementation part of the Baseline+ extended feature extraction methods is shown below:

The notations used are ‘st’ for stack, ‘buf’ for buffer, ‘top’ for stack top and buffer head, ‘2^{nd} top’ for second element in stack or buffer if available, ‘3^{rd} top’ for third element in stack of buffer if available, ‘val’ for valency, ‘POS’ for Part Of Speech tag of the given element in stack of buffer.

The Baseline+ model features are same as the Baseline model + other extracted features, which are used or skipped while creating their combinations and evaluating performances.

For finding the valency, we used the size of the left or right context of an element, and is found by adding the child element of a node of the Dependency tree.
’

We created a class Parse for finding the valency of a word, from its size of dependents.

```
class Parse(object):
    def __init__(self,size):
        self.size = size
        self.heads = [None]*(size + 1)
        self.left_arc = []
        self.right_arc = []

        for i in range(size+1):
            self.left_arc.append(defaultList(0))
            self.right_arc.append(defaultList(0))

    def add_arc(self,head,child):
        self.heads[child] = head
        if child < head:
            self.left_arc[head].append(child)
        else:
            self.right_arc[head].append(child)
```

The function shown below finds the valency of the word of a context type (word of POS tag), from its subtree.

```
def   get_node_children(word,data,subtree):   if
          word == -1 :
              return 0,'',''
          dependents = subtree[word] valency =
          len(dependents)

          if   not   valency:   return
              0,'',''

          elif valency == 1:
              return 1,data[dependents[-1]],''
          else:
              return valency,data[dependents[-1]],data[dependents[-2]]
```

The different features extracted are shown below :

```
word_st_top,word_st_2ndtop,word_st_3rdtop = get_stackcontext_features('words',stack)
POS_st_top,POS_st_2ndtop,POS_st_3rdtop = get_stackcontext_features('POS_tags',stack)


word_buf_top,word_buf_2ndtop,word_buf_3rdtop = get_buffercontext_features('words',buffer)
POS_buf_top,POS_buf_2ndtop,POS_buf_3rdtop = get_buffercontext_features('POS_tags',buffer)


bufleft_word_val,bufleft_word_child1,bufleft_word_child2 = get_node_children(curr_word,tokens_dict,
                                                           tree_dependencies.left_arc)
bufleft_tag_val,bufleft_tag_child1,bufleft_tag_child2 = get_node_children(curr_word,POS_tags,
                                                           tree_dependencies.left_arc)

bufright_word_val,bufright_word_child1,bufright_word_child2 = get_node_children(curr_word,tokens_dict,
                                                           tree_dependencies.right_arc)
_,bufright_tag_child1,bufright_tag_child2 = get_node_children(curr_word,POS_tags,
                                                           tree_dependencies.right_arc)


stleft_word_val,stleft_word_child1,stleft_word_child2 = get_node_children(stack_top,tokens_dict,
                                                           tree_dependencies.left_arc)
_,stleft_tag_child1,stleft_tag_child2 = get_node_children(stack_top,POS_tags,
                                                           tree_dependencies.left_arc)


stright_word_val,stright_word_child1,stright_word_child2 = get_node_children(stack_top,tokens_dict,
                                                           tree_dependencies.right_arc)
_,stright_tag_child1,stright_tag_child2 = get_node_children(stack_top,POS_tags,
                                                           tree_dependencies.right_arc)

distance = 0

if stack_top != 0 and curr_word != -1:
    distance  = min(curr_word - stack_top,5)
```

Figure 9: Various features used for Baseline+ Model.

The above snippet shows the various features we extracted from the data, these features were then used in combination to create other feature sets, and the model's performance was noted.

The various feature sets created using the above extracted elementary features are shown in the next page.

```
        context_features['word_tag_pairs'] = (
            (word_st_top,POS_st_top),
            (word_buf_top,POS_buf_top),
            (word_buf_2ndtop,POS_buf_2ndtop),
            (word_buf_3rdtop,POS_buf_3rdtop)
           )

        context_features['bigram_features'] = (
            (word_st_top,word_buf_top),
            (POS_st_top,POS_buf_top),
            (POS_buf_top,POS_buf_2ndtop),
        )

        context_features['trigram_features'] = (
            (POS_buf_top,POS_buf_2ndtop,POS_buf_3rdtop),
            (POS_st_top,POS_buf_top,POS_buf_2ndtop),
            (POS_st_top,POS_st_2ndtop,POS_buf_top),
            (POS_st_top,stright_tag_child1,POS_buf_top),
            (POS_st_top,POS_buf_top,bufleft_tag_child1),
            (POS_st_top,stleft_tag_child1,stleft_tag_child2),
            (POS_st_top,stright_tag_child1,stright_tag_child2),
            (POS_buf_top,bufleft_tag_child1,bufleft_tag_child2),
            (POS_st_top,POS_st_2ndtop,POS_st_3rdtop)
                           )

        context_features['word_val_feat'] = (
            (word_st_top,stright_word_val),
            (word_st_top,stleft_word_val),
            (word_buf_top,bufleft_word_val)
                        )

        context_features['tag_val_feat'] = (
            (POS_st_top,stright_word_val),
            (POS_st_top,stleft_word_val),
            (POS_buf_top,bufleft_word_val)
                        )

        context_features['distance_feat'] = (
            (word_st_top,distance),
            (word_buf_top,distance),
            (POS_st_top,distance),
            (POS_buf_top,distance),
            ('tag '+ POS_st_top + POS_buf_top,distance),
            ('word_ft ' + word_st_top + word_buf_top,distance)
        )
```

After creating the features and label sets, they look like the following :

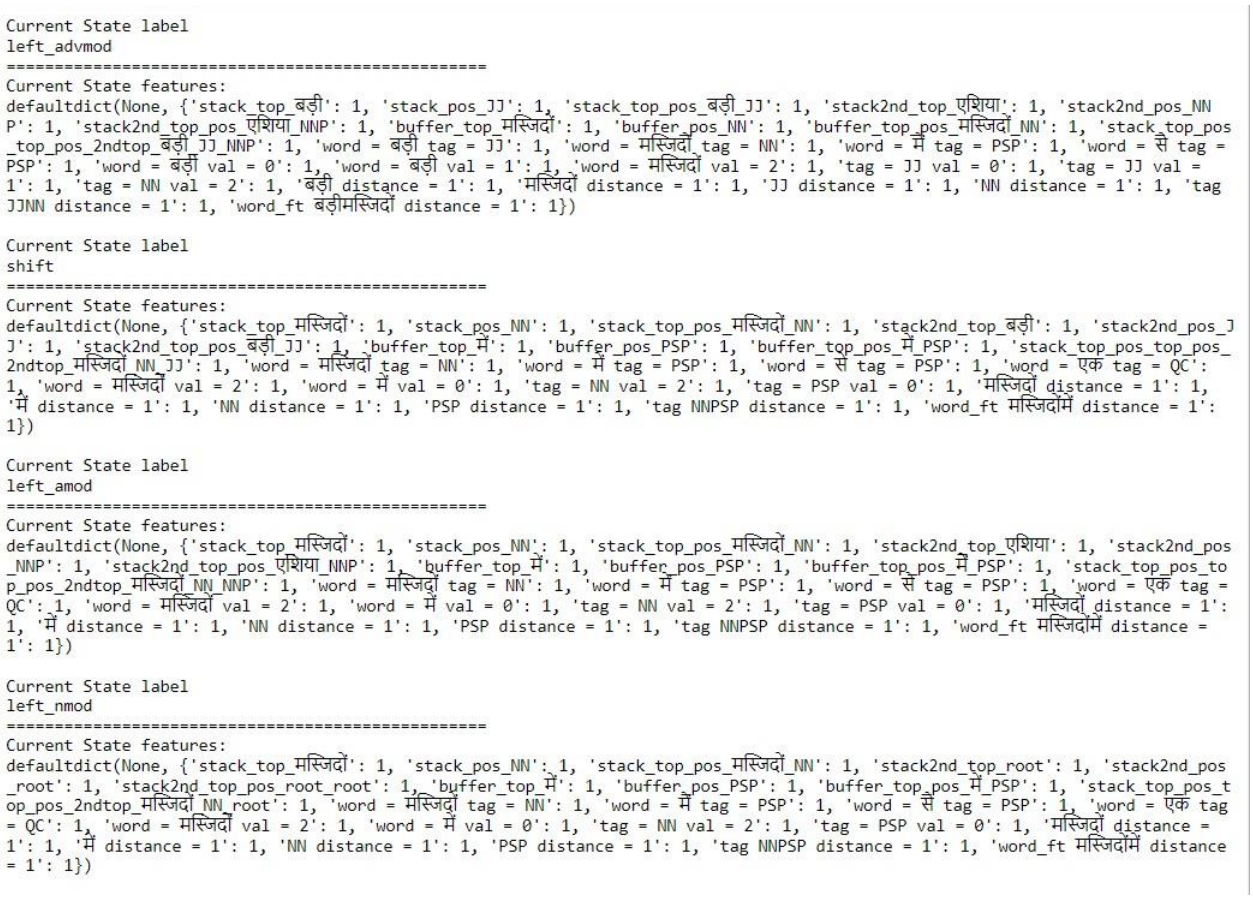

Figure 10: The final look of the features and labels.

Since there are unique combinations of a high number of features, the total number of features was very high for the set of features which gave us the best model, and hence the dimensionality of training dataset was

(Number of unique states(Buffer, Stack and Arc) x Number of unique features)

We one hot encoded the features, whenever a feature was present in the current state, as a result the size of the dataset was very high, which is shown below.

The size of this matrix is so high that it'd require 3313.331 GB of memory, even though the number of elements present (i.e 1) is very low, the metric to measure this phenomenon is called a matrix's sparsity.

```
x_train
```

```
<460986x112304 sparse matrix of type '<class 'numpy.float64'>'
        with 10460145 stored elements in LInked List format>
```

Here, it is shown below,

```
num_elements_present = 10460145
size_of_matrix = 460986 * 112304

sparsity = num_elements_present/ size_of_matrix

sparsity
```

```
0.00020204808731347047
```

The baseline model also had high dimensionality, but this new feature set increased the size considerably.

```
size_of_old_matrix = 460986 * 50532

size_of_new_matrix = 460986 * 112304

size_of_new_matrix/size_of_old_matrix
```

```
2.222433309586005
```

The new features caused the dataset to be 2.22 times the older one.

So, it was reasonable to convert the matrix to sparse format. And to train a machine learning model, like previous case, it was required to use a classifier which supports sparse matrix as an input.

In the Baseline analysis, we established that Logistic Regression classifier outperformed Naïve Bayes, so we went with the same model for Baseline+ as well.

3 Metrics and Results

We need to evaluate our classifiers which we discussed in the previous part, for that the choice of evaluation metrics are LAS and UAS, which works in a same way as accuracy but also attaches some semantic meaning to it.

1. **LAS (Labelled Attachment Score):**
   The labelled attachment score (LAS) evaluates the output of a parser by considering how many words have been assigned both the correct syntactic head and the correct label (dependency relation eg. `subj`). If parse trees and gold standard trees can be assumed to have the same yield, and if no syntactic relations are excluded, then it reduces to a simple accuracy score, but in general it can be defined as the labelled F1-score of syntactic relations. It associates semantic meaning to the score.

   In simple words:
   `Labelled attachment score (LAS) = Percentage of words that get the correct head and label`

2. **UAS (Un-Labelled Attachment Score):**
   It evaluates how many words have correctly been assigned correct syntactic head, it has no semantic meaning attached to it.

   In simple words:
   `Unlabelled attachment score (UAS) = Percentage of words that get the correct head`

For this, we wrote our own custom code in python, for each sentence there are dependency trees, and for each dependency tree we calculate the score depending on the number of children it has and is added with the next set of sentences, a small snippet of our implementation is shown below:

```
        unlabelled_attachment_score = 0
        labelled_attachment_score = 0
        total_children = 0

        for child in test_dependency_tree: token_head,arc_label
            = test_dependency_tree[child]

            if token_head == test_heads[child]:
                unlabelled_attachment_score += 1
                if arc_label ==
                test_labels[child]:
                    labelled_attachment_score += 1 total_children += 1 return
        [unlabelled_attachment_score,labelled_attachment_score,total_children]
```

3.1 Results

We trained on the train CoNLL-U file and tested on the test CoNLL-U file, and obtained the following results:

3.2 Results on Baseline Model

Classifier Name	**Unlabelled Attachment Score**	**Labelled Attachment Score**
Bernoulli Naive Bayes	76.4	66.9
Logistic regression with L2penalty	84.1	75.2
Logistic regression with saga solver + Elastic Net penalty (L1-ratio=0.25)	85.2	75.93

Also, the time taken by various classifiers to train them varied significantly, from the fastest being Bernoulli Naïve Bayes to the slowest begin Logistic regression with Elastic Net penalty.

Classifier Name	**Time to Train**
Bernoulli Naive Bayes	1.5 Seconds
Logistic regression with L2penalty	157 seconds (2 min 37 seconds)

Logistic regression with saga solver + Elastic Net penalty (L1ratio=0.25)	25,378 seconds (7+ hours)

We see that the best results are obtained by the Logistic Regression Classifier with Elastic Net penalty, even though the improvement over simple Logistic Regression with L2-penalty is very less, it takes a significantly longer time to train (around 160 times longer), so the time-performance trade-off is not that fruitful, though it is significantly better than the Naïve Bayes classifier.

3.2.1 Results on Baseline+ Model

Now we show the difference in performance after including various combinations of feature sets.

The following table shows results for **Logistic Regression with L2-penalty**.

Feature Set	**Unlabelled Attachment Score**	**Labelled Attachment Score**
Baseline Features	84.1	75.2
Baseline Features + wordtag pairs	84.45	75.26
Baseline Features + bigram features	83.9	74.15
Baseline Features + bigram features + trigram features + word-tag pairs	84.8	75.23
Baseline Features + wordtag-pairs + trigram features	84.9	75.35
Baseline Features + wordtag-pairs + valency features	85.2	76.33

Baseline Features + wordtag-pairs + valency features + trigram features + bigram features	84.9	75.28
Baseline Features + trigram features + bigram features + word-tag-pairs + valency features + distance features	85.22	75.7
Baseline Features + valency features + word-tag pairs + distance features	86.355	77.236
Baseline Features + valency features + distance features	86.405	77.335

We also tested the performance of different classifiers.

Classifier Name	**Unlabelled Attachment Score**	**Labelled Attachment Score**
Bernoulli Naive Bayes	73.15	63.23
Logistic regression with L2penalty	86.405	77.335
Logistic regression with saga solver + Elastic Net penalty (L1-ratio=0.25)	86.88	77.412

Also, the time taken by various classifiers to train them varied significantly, from the fastest being Bernoulli Naïve Bayes to the slowest begin Logistic regression with Elastic Net penalty (a combination of both L1 and L2 loss for which is better sparse indices and reduces noise on the data).

Classifier Name	**Time to Train**
Bernoulli Naive Bayes	3.35 seconds

Logistic regression with L2penalty	233 seconds (3 min 53 seconds)
Logistic regression with saga solver + Elastic Net penalty (L1ratio=0.25)	44,158 seconds (12+ hours)

4 Conclusion and Future Works

We see that the best result is shown by only keeping valency and distance features, in the paper by Zhang and Nivre [1], they got best results by including all the context features together, but they did their research for English and Chinese Language, whereas our project was conducted for Hindi Language, which differs from the above two languages a bit in structure and formation, so it might be possible that the context captured may be ambiguous, for example the word 'कल' in hindi can mean both yesterday or tomorrow, so in order to capture this context information, maybe more training data will be required or some advanced set of features should be manually provided in the dataset.

As a project for future work, we can try out this project for more advanced Neural Based Models, Attention Based models or graphical models.

References

[1] Yue Zhang and Jaokim Nivre
"Transition-based Dependency Parsing with Rich Non-local Features"

[2] Dr. Pawan Goyal
NLP Lecture Series on NPTEL Platform