Computational Linguistics Project 2019-20

IIIT-Hyderabad

Co-reference Resolution (Hindi)

By Akshat Gahoi (2018114012), Akshat Chhajer (2018114008)

https://github.com/akshatcx/coreference_resolution_cl

Acknowledgements

We like to thank Dr. Dipti Misra Sharma to give us this project and our mentor Mr. Anirudh Dahiya and LTRC, IIIT Hydeabad for their support.

Abstract

Informally, Co-reference resolution is the task of finding all expressions that refer to the same entity in a text.

Formally, co-reference consists of two linguistic expressions—antecedent and anaphor. The anaphor is the expression whose interpretation (i.e., associating it with an either concrete or abstract real-world entity) depends on that of the other expression. The antecedent is the linguistic expression on which an anaphor depends. Thus, the coreference resolution task is to discover the antecedent for each anaphor in a given text.

It is an important step for a lot of higher level NLP tasks that involve natural language understanding such as document summarization, question answering, and information extraction.

Research on coreference resolution in the general English domain dates back to 1960s and 1970s. However, research on coreference resolution in the field of indian regional languages like hindi has not seen major development.

Project Outline

1. Data Collection:

For this project we will be using the data from the ‘Hindi/Urdu Dependency Treebank’ (Bhatt et al., 2009) (http://ltrc.iiit.ac.in/hutb_release/). It is a rich corpus with various linguistic information. The annotation is based on Computational Paninian Grammar 977 (CPG-henceforth) framework.

2. **Pre-Processing:**

After doing some research, we concluded that dependency trees would be the most efficient and effective way to represent our data-set. Thus, we aim to pre-process the data in python by converting the raw text into specific data structures which can be used to easily implement dependency trees. This will be done using the SSP API for python (https://readthedocs.org/projects/ssf-api-python).
The dependency trees obtained will act as the core for the further research in this project due to their extreme relevance and usability.

3. Main Approach:

Our main approach to resolve the problem statement will be a rule based approach. This module will attempt to resolve the pronoun, using the dependency relations and other information, based on the category of the pronoun, which is decided using an exhaustive list of pronoun categories. Some of the various categories based on which our rule based system will work on are:
1. Reflexives
2. Place Pronouns

3. Relative Pronouns
4. Personal Pronouns

4. Result and Error Analysis:

After successfully doing the required task, we will carefully look at the results and will do a comparative analysis. After this a thorough error analysis will also be done which will hopefully give us some insight as to where the improvements can be done.

Project

Firstly, the ssf_api was used to convert the annotated data into dependency trees. It took raw data and genrerated an object with the following attributes for each datafile:

1) SSF Info
2) Node
3) Sentence
4) Chunk
5) Word
6) Feature Set

ssf_api.py

```
57          self.wordNumList.append(word)
58
59      def addNode(self,node):
60          print self.sentenceNum ,node.nodeName
61          self.nodeDict[node.nodeName]=node
62
63      def addChunkNumToNode(self,node_name,chunk_num):
64          self.nodeDict[node_name].chunkNum=chunk_num
65
66  class Chunk():
67      def __init__(self,tag,node_name,features_set,sentenceNum,chunk_num):
68          # print "-"*30,"new chunk with tag - %s and f = %s"%(tag,features_string)
69          self.chunkTag=tag
70          self.chunkNum=chunk_num
71          self.nodeName=node_name
72          self.featureSet=features_set
73          self.wordNumList=[]
74          self.sentenceNum=sentenceNum
75
76      def addWord(self,word):
77          # print "-"*30,"adding word to chunk"
78          self.wordNumList.append(word)
79
80  class Word():
81
82      def __init__(self,word,tag,features_string,extra_features,sentenceNum,chunkNum):
83          # print "-"*30,"new word- %s with tag - %s and f = %s"%(word,tag,features_string)
84          self.wordTag=tag;
85          self.word=word.decode("utf-8")
86          #Have modified this, wont work in generic case
87          self.featureSet=extra_features
88          self.extraFeatureSet=extra_features
89          self.sentenceNum=sentenceNum
90          self.chunkNum=chunkNum
91          self.sense=None
92          self.conn=False
93          self.splitConn=False
94          self.arg1=False
95          self.arg2=False
96          self.relationNum=None
97          self.arg1Span=None
98          self.arg2Span=None
```

Information about the chunks, sentence, tags and relation between different nodes then can be easily extracted and used.

After running ssf_api.py we had a dataset with dependency trees. Now, a rule based hybrid approach using dependency trees was implemented to resolve the coreferences.

Hybrid approach using dependency

Most coreference resolution approaches which use deeper linguistic features are based on phrase structure based grammatical models. We, instead, use linguistic features from a dependency grammar model . Hence, our first aim is to develop a rule based approach by studying and analyzing patterns in the CPG based dependency structure that can be formulated as rules to resolve references. Our observation shows that some categories of pronominal references such as Reflexives, Relatives, First and Second person can be easily resolved by formulating rules based on dependency structures.

However, for some ambiguous references, specifically in third person pronouns, although syntactic constraint do reduce search space, they fail to uniquely identify the referent.In resolution of such references, morphological, grammatical, distance and semantic features also play an important role. We have to implement a supervised learning algorithm using these features which was beyond our scope for now simply because it required a proper annotated data with all the features which was not there because of non-availability of a proper indigenous morph-anaylser.

Rule based resolution module

The rule based module attempts to locate the referent of the pronoun using the constraints derived from the dependency structures and relations. First, the category of the pronoun is identified. After the category is identified, a set of rules defined for each category is applied to locate the referent. We discuss below these pronominal forms and the rules derived for their resolution:

1. Reflexives :

The rules for resolution of reflexive pronouns can be consolidated in the following algorithm:

• If a node ‘N’ is a reflexive pronoun, then move up in the tree, until an NP node or a verb (VGF) node is encountered. Call this node ‘X’.

• If ‘X’ is NP, then search for children nodes of ‘X’ other than ‘N’, if such a children node with dependency label ‘r6’ is found, propose it as referent. otherwise to go Next step.

• If ‘X’ is verb node, then search in the children of the node ‘X’ with dependency label ‘k1’, other than ‘N’, if such a node is found, propose it a referent, else traverse up in the tree.

• If ‘X’ is a CCP (conjunct) node with label ‘k1’, search in the children of ‘X’, propose all the NP nodes in the children set as referent.

• If ‘X’ is root of tree and no referent is found, then output NULL (no solution)

```
    for _ in sc:
        s = var.sentenceList[_[0]]
        c = s.chunkList[_[1]]
        n = s.nodeDict[c.nodeName]
        x = s.nodeDict[n.nodeParent]
        fans={}
        fans['sentence'] = _[0]
        fans['pronoun'] = _[1]
        fans['referents'] = []
        flag = 1
        while(flag):
            if x.nodeName[:2] == 'NP':
                for child in x.childList:
                    if s.nodeDict[child].nodeRelation == 'r6' and s.nodeDict[child].nodeName != n.nodeName:
                        fans['referents'].append(s.nodeDict[child].chunkNum)
                        flag = 0
            elif x.nodeName[:3] == 'VGF':
                for child in x.childList:
                    if s.nodeDict[child].nodeRelation == 'k1' and s.nodeDict[child].nodeName != n.nodeName:
                        fans['referents'].append(s.nodeDict[child].chunkNum)
                        flag = 0
            elif x.nodeName[:3] == 'CCP' and x.nodeRelation == 'k1':
                for child in x.childList:
                    if s.nodeDict[child].nodeName[:2] == 'NP' and s.nodeDict[child].nodeName != n.nodeName:
                        fans['referents'].append(s.nodeDict[child].chunkNum)
                        flag = 0
            if x.nodeParent == "None":
                break
            x = s.nodeDict[x.nodeParent]
        res.append(fans)

```

2. Spatial :

Following algorithm can be derived for resolution of spatial pronouns:

• Begin with pronoun identified as ‘spatial’ (or place pronoun).

• Search in reverse order for a noun phrase with Named Entity Category as ‘LOCATION’ and dependency label ‘k7p’

• If such a noun phrase is found, predict it as the referent.

• Else Search in reverse order only for a noun phrase with dependent label ‘k7p’ up-to 3 sentence.

• If such a noun phrase is found, predict it as the referent.

• Else Search in reverse order only for a noun phrase with Named Entity Category as ‘LOCATION’ up-to 3 sentence.

• If such a noun phrase is found, predict it as the referent.

• If no referent is found within 3 sentences, output ‘NULL’ (no solution).

```
            if word.word in li:
                sc.append([word.sentenceNum, word.chunkNum])
    for _ in sc:
        s = var.sentenceList[_[0]]
        c = s.chunkList[_[1]]
        n = s.nodeDict[c.nodeName]
        fans={}
        fans['sentence'] = _[0]
        fans['pronoun'] = _[1]
        fans['referents'] = []
        flag = 1
        limit=3
        while(flag and limit):
            for chunk in s.chunkList:
                if chunk.nodeName[:2] == 'NP' and s.nodeDict[chunk.nodeName].nodeRelation == 'k7p' and s.nodeDict[chunk.nodeName] != n:
                    fans['referents'].append(chunk.chunkNum)
                    flag=0
                    break
            try:
                s = var.sentenceList[_[0]-1]
                limit=limit-1
            except:
                break
        res.append(fans)
```

3. Relative :

Following algorithm can be derived for resolution of relative pronouns:

• Start with the pronoun node ‘N’, move up in the tree until a verb node (VGF or VGNF) is encountered. Call this node as ‘X’.

• If the label of attachment of the ‘X’ node with its parent node (‘H’) is ‘nmod-relc’, then select the parent node (‘H’) as the referent. Else, keep moving upwards until such a node is found.

• If the ‘X’ is the root of the tree and no referent is found, then output ‘NULL’ (no solution

```
5               sc.append([word.sentenceNum, word.chunkNum])
6       for _ in sc:
7           s = var.sentenceList[_[0]]
8           c = s.chunkList[_[1]]
9           n = s.nodeDict[c.nodeName]
10          x = s.nodeDict[n.nodeParent]
11          fans={}
12          fans['sentence'] = _[0]
13          fans['pronoun'] = _[1]
14          fans['referents'] = []
15          flag = 1
16          while(flag):
17              if x.nodeName[:3] == 'VGF' or x.nodeName[:4] == 'VGNF':
18                  if x.nodeRelation == 'nmod__relc':
19                      fans['referents'].append(s.nodeDict[x.nodeParent].chunkNum)
20                      flag = 0
21              if x.nodeParent == "None":
22                  break
23              x = s.nodeDict[x.nodeParent]
24          res.append(fans)
25
```

4. First Person :

Following algorithms can be derived for resolution of first person pronouns:

• Starting at the pronoun node, move up in the tree until a complementizer node kF (‘that’) is encountered. Call this node ‘C’.

• If no such node is found up-to the root, output ‘no solution’

• Move up to the parent node of ‘C’. If this is a verb node. call this ‘V’, else output ‘no solution’.

• Search in the children nodes of ‘V’, nodes with label ‘k1’ and ‘k4’.

• For first person pronoun, select node with ‘k1’ as the referent, If no such nodes are found, output ‘no solution’.

```
    for word in var.globalWordList:
        if word.word in li:
            sc.append([word.sentenceNum, word.chunkNum])
    for _ in sc:
        s = var.sentenceList[_[0]]
        c = s.chunkList[_[1]]
        n = s.nodeDict[c.nodeName]
        x = s.nodeDict[n.nodeParent]
        fans={}
        fans['sentence'] = _[0]
        fans['pronoun'] = _[1]
        fans['referents'] = []
        flag = 1
        while(flag):
            if var.globalWordList[s.chunkList[x.chunkNum].wordNumList[0]].word == complementizer:
                x = s.nodeDict[x.nodeParent]
                if x.nodeName[:1] == 'V':
                    for child in x.childList:
                        if s.nodeDict[child].nodeRelation == 'k1':
                            fans['referents'].append(s.nodeDict[child].chunkNum)
                            flag = 0
                else:
                    break
            if x.nodeParent == "None":
                break
            x = s.nodeDict[x.nodeParent]
        res.append(fans)
```

5. Second Person :

Following algorithms can be derived for resolution of second person pronouns:

• Starting at the pronoun node, move up in the tree until a complementizer node kF (‘that’) is encountered. Call this node ‘C’.

• If no such node is found up-to the root, output ‘no solution’

• Move up to the parent node of ‘C’. If this is a verb node. call this ‘V’, else output ‘no solution’.

• Search in the children nodes of ‘V’, nodes with label ‘k1’ and ‘k4’.

• For second person pronoun, select node with ‘k4’ as the referent. If no such nodes are found, output ‘no solution’.

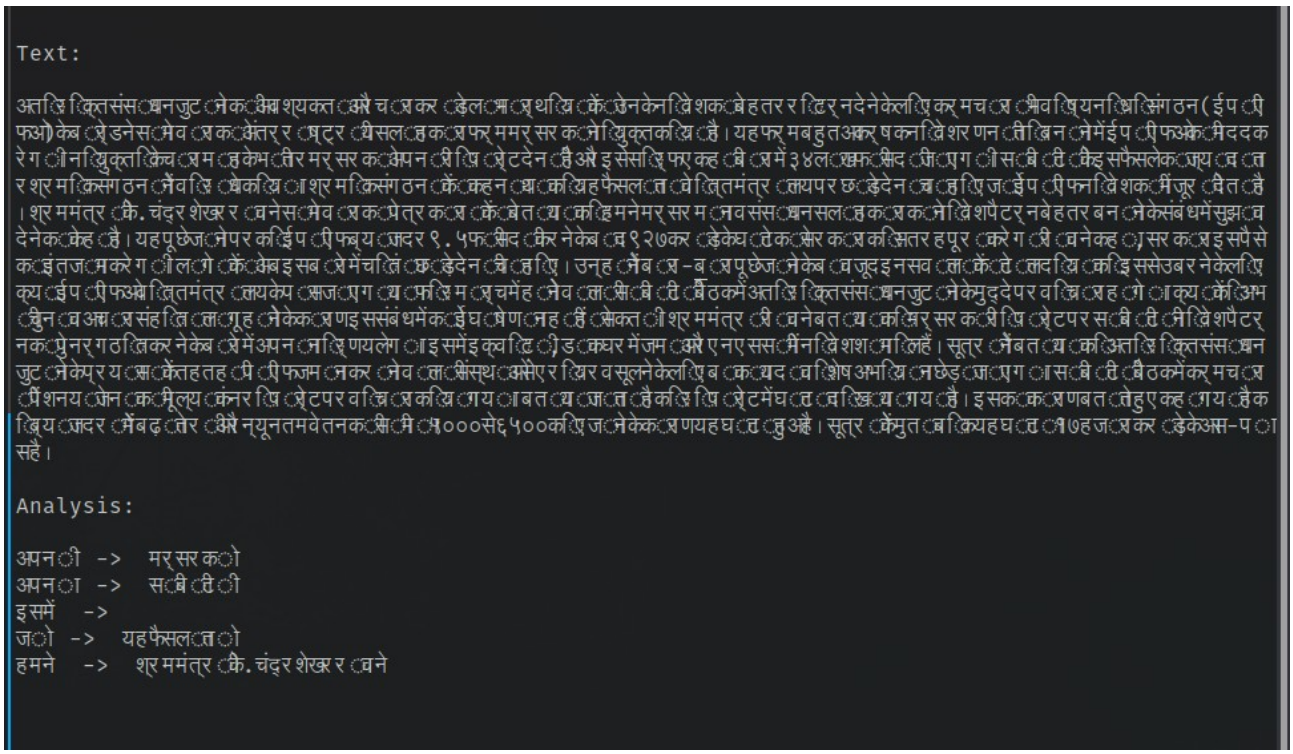
```
def r5(var,res,li):
    sc=[]
    for word in var.globalWordList:
        if word.word in li:
            sc.append([word.sentenceNum, word.chunkNum])
    for _ in sc:
        s = var.sentenceList[_[0]]
        c = s.chunkList[_[1]]
        n = s.nodeDict[c.nodeName]
        x = s.nodeDict[n.nodeParent]
        fans={}
        fans['sentence'] = _[0]
        fans['pronoun'] = _[1]
        fans['referents'] = []
        flag = 1
        while(flag):
            if var.globalWordList[s.chunkList[x.chunkNum].wordNumList[0]].word == complementizer:
                x = s.nodeDict[x.nodeParent]
                if x.nodeName[:1] == 'V':
                    for child in x.childList:
                        if s.nodeDict[child].nodeRelation == 'k4':
                            fans['referents'].append(s.nodeDict[child].chunkNum)
                            flag = 0
                else:
                    break
            if x.nodeParent == "None":
                break
            x = s.nodeDict[x.nodeParent]
        res.append(fans)
```

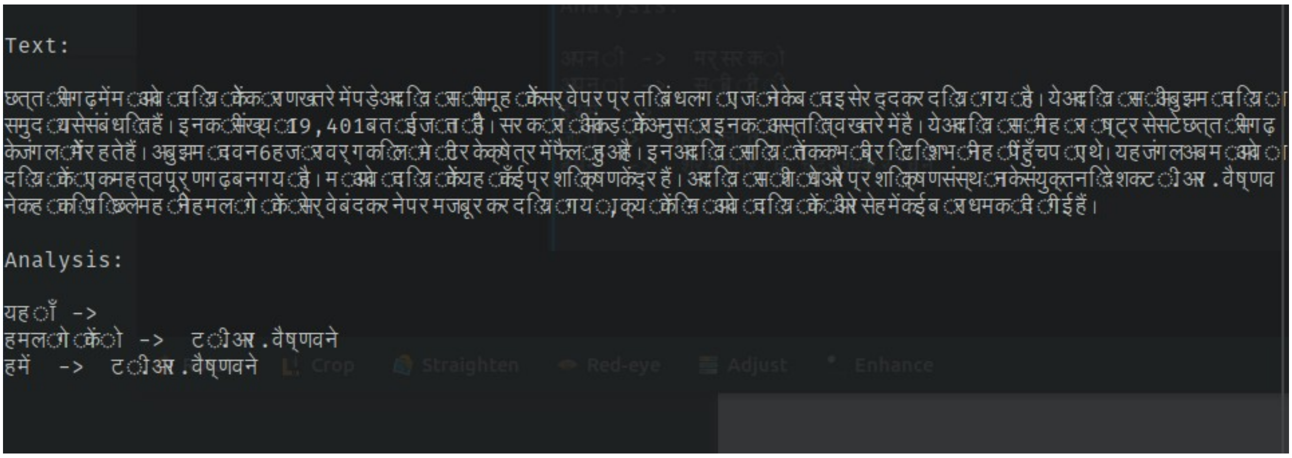
Implementation of Code and Output Format

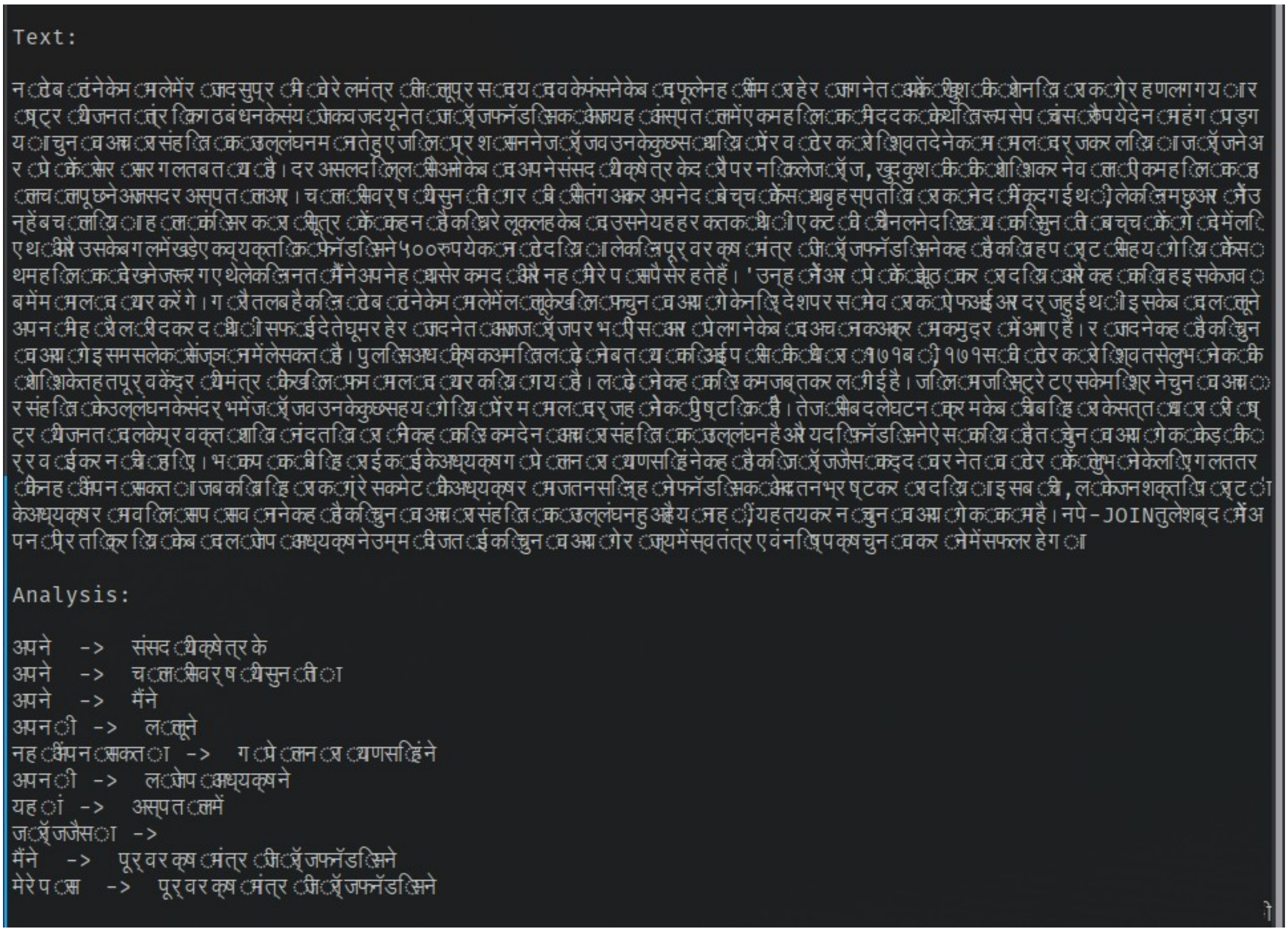
We imlemented the above given rules for our data and made a file “coref.py” which check through all the rules in our data give respective anaphora resolution.

In the output we will have our data with its respective analysis in which each pronoun is matched with every word it is refering to.

Output Examples:







Accuracy

68.11%