1.Initial Tagger:

Initial Tagger.py

__init__.py

->Initial Tagger contains two functions

i) initializeSentence(FREDICT, sentence) return taggedSentence

ii)initializeCorpus(FREDICT, inputFile, Outfile)

The FREDICT here is the sdict,or short lexicon dictionary created from the gold corpus. A word and a tag is imputed in sdict, if the word occurred more than 1 time. If a word is associated with multiple tags, then the tag with most frequency is added to the sdict.

The above two functions is used to tag the raw corpus generated from the gold corpus and to form a intialtaggedFile.

Steps performed in initializeSentence(FREDICT, sentence):

1.if word in FREQDICT:

```
tag = FREQDICT[word]
```

2. elif lowerW in FREQDICT:

```
tag = FREQDICT[lowerW]
```

3. Then check in which category does the word comes in Number, Capital, Unknown. ("TAG4UNKN-NUM", "TAG4UNKN-CAPITAL", "TAG4UNKN-WORD")

2.Utility

Config.py

Eval.py

LexiconCreator.py

Utils.py

__init__.py

Config.py file contains the code:>

```
#Change the value of NUMBER_OF_PROCESSES to obtain faster tagging process!
NUMBER_OF_PROCESSES = 2
```

THRESHOLD = (3, 2)

Eval.py contains two functions
i) def computeAccuracy(goldStandardCorpus, taggedCorpus):
Return accuracy
ii) computeAccuracies(fullDictFile, goldStandardCorpus, taggedCorpus):
Return knownwors_accuracy, unknownwords_accuracy, overall_accuracy

Utils.py contains 3 utility functions:>
i) getWordTag(wordTag):
Return word, tag

ii) getRawText(inputFile, outFile):
From a tagged file, outputs a raw text file

iii) readDictionary(inputFile):
Return dictionary
Reads the .dict or .sdict file and returns a word tag stored in dictionary data type

LexiconCreator.py contains two functions:>
i)add2WordTagFreqDict(word, tag, inDict):

```
if word not in inDict:
    inDict[word] = {}
    inDict[word][tag] = 1
else:
    if tag not in inDict[word]:
        inDict[word][tag] = 1
    else:
        inDict[word][tag] += 1
```

- A word might be associated more than 1 tag,this snippet track the frequency of word associated with every tag involved with.

ii)createLexicon(corpusFilePath, fullLexicon):
corpusFilePath = Gold Standard Path,

fullLexicon = ‘full’ or ‘short’ , full argument want the function to create a .dict file and short argument wants the function to create a .sdict file.

Steps in createLexicon:>
1.getWordTagCounter: this is a dictionary containing a dictionary such as {'༄༅།_།': {'P': 1}, 'རྒྱལ་པོ་': {'P': 29}, 'ལ་': {'P': 243, 'C': 21}, 'གཏམ་': {'P': 11}, 'བྱ་བ་': {'P': 21}, 'རིན་པོ་ཆེ་-འི་': {'P': 8}, 'ཕྲེང་བ': {'P': 2}, '།_༄༅།_།': {'P': 5}, 'རྒྱ་གར་': {'P': 8}, 'སྐད་': {'N': 3, 'P': 2}, 'དུ': {'C': 10, 'P': 41}, '།_': {'P': 26}, 'རཱ་': {'P': 1}, 'ཛ་པ་': {'P': 1}, ...}
Notice some words are associated with more than 1 tag.

2.After this if fullLexicon == ‘full’:
  The dictionary word tag pair frequency is sorted and most frequent is stored.
 Elif fullLexicon == ‘short’:
  If a word occurs more than once, then it is stored.

3.Finally is file is written in .dict or .sDict file.

3.pSCRDRtagger
- RDRPOSTagger.py
- ExtRDRPOSTagger.py
- __init__.py

class named RDRPOSTagger that extends or inherits from another class called SCRDRTree.

RDRPOSTagger.py is responsible for training and tagging. From training, a .dict file and .rdr file is produced from the gold corpus.
Steps in Training the file:>>>
1.Create .dict and .sDict file or full and short dictionary files from gold standard corpus
2.Extract the raw corpus file from gold standard corpus as .RAW
3.Generate a initial tagged file as .INIT on raw corpus using .sdict short dictionary file (if the words are not there, then tag involved with "TAG4UNKN-NUM", "TAG4UNKN-CAPITAL", "TAG4UNKN-WORD" are used.

4. Using the gold standard corpus and threshold values, the function SCRDRTreeLearner generates rdr rules as .RDR files.
5. Then .sDict , .RAW and .INIT files are deleted.
6.Only .RDR and .DICT remains.

Steps in Tagging the file:>

1. Reads model from .RDR using class method constructSCRDRtreeFromRDRfile.
2. Then reads the full lexicon dictionary .Dict
3. Using function tagRawCorpus, its tag the raw text as .TAGGED file.
   Important code snippet:>

```
def tagRawSentence(self, DICT, rawLine):
        line = initializeSentence(DICT, rawLine)
        sen = []
        wordTags = line.split()
        for i in range(len(wordTags)):
            fwObject = FWObject.getFWObject(wordTags, i)
            word, tag = getWordTag(wordTags[i])
            node = self.findFiredNode(fwObject)
            if node.depth > 0:
                sen.append(word + "/" + node.conclusion)
            else:# Fired at root, return initialized tag
                sen.append(word + "/" + tag)
        return " ".join(sen)
```

4.SCRDRlearner

- Node.py
- Object.py
- SCRDRTree.py
- SCRDRTreeLearner.py
- __init__.py

SCRDR full form is Separation of Context and Reasoning in Decision Rules

Object.py :>

Defined a class name Object with attributes such as ["word",
"tag",
"prevWord2",
"prevWord1",
"nextWord1",
"nextWord2",
"prevTag2",
"prevTag1",
"nextTag1",
"nextTag2",
"suffixL2",
"suffixL3",
"suffixL4"].

Contains function such as

i) def getObject(wordTags, index):#Sequence of "Word/Tag"
return Object(word, tag, preWord2, preWord1, nextWord1, nextWord2, preTag2, preTag1, nextTag1, nextTag2, suffixL2, suffixL3, suffixL4)

ii) def getObjectDictionary(initializedCorpus, goldStandardCorpus):

Important code snippet:

```
if initTag not in objects.keys():
    objects[initTag] = {}
    objects[initTag][initTag] = []

if correctTag not in objects[initTag].keys():
    objects[initTag][correctTag] = []

objects[initTag][correctTag].append(getObject(initWordTags, k))

return objects
```

*In function getObjectDictionary, it takes gold corpus and initializedCorpus and then returns a dictionary type name ‘objects’ with the help from getObject function.

Return variable objects would look something like this:>

```
Objects = {
                ‘P’:{‘P’:[...], ‘C’:[...]},
                ‘N’:{‘N’:[...]},
                …
}
```
[...] contains values of word, tag, preWord2, preWord1, nextWord1, nextWord2, preTag2, preTag1, nextTag1, nextTag2, suffixL2, suffixL3, suffixL4.

SCRDRTreeLearner.py:>
i)def findMostEfficientRule(self, startTag, objects, correctCounts):

A code snippet:>

```
for tag in objects:
        if tag == startTag:
            continue
        if len(objects[tag]) <= maxImp or len(objects[tag]) <
self.improvedThreshold:
            continue
```

Explanation: if the tag == startTag(gold standard tag), then continue. And if the wrong tag occurred is less than self.improvedThreshold (the first threshold parameter), then continue. If those two above if condition is met, then no rules would be generated.

ii)#For layer-2 exception structure ---------->

```
   def findMostImprovingRuleForTag(self, startTag, correctTag, correctCounts,
wrongObjects):
        impCounts, affectedObjects = countMatching(wrongObjects, [])

        maxImp = -1000000
        bestRule = ""
        for rule in impCounts:
            temp = impCounts[rule]
            if rule in correctCounts:
                temp -= correctCounts[rule]
```

```
            if temp > maxImp:
                maxImp = temp
                bestRule = rule

        if maxImp == -1000000:
            affectedObjects[bestRule] = []

        return bestRule, maxImp, affectedObjects[bestRule]
```

Explanation:> The above function is used to find the best rule, and they find the best rule by selecting the most improving rule for Tag. countMatching returns object that are incorrectly labeled and their possible rules with frequency. They then in a loop, check if the particular rule is present in correctCounts which stores rules frequency of correctly labeled words. If present, their frequency is subtracted and the rule with the largest frequency difference left is considered the best rule.