```
from collections import defaultdict

pos = defaultdict(list)
pos['sleep'] = ['NOUN', 'VERB']
pos['ideas']  # Accessing a key that doesn't exist defaults to an empty list
# Output: []
```

```
[]
```

```
from collections import defaultdict

pos = defaultdict(lambda: 'NOUN')  # [1]
pos['colorless'] = 'ADJ'
pos['blog']  # [2]
# Output: 'NOUN'
```

```
'NOUN'
```

```
list(pos.items())
```

```
[('colorless', 'ADJ'), ('blog', 'NOUN')]
```

```
import nltk
from collections import defaultdict

# Load the text and create a frequency distribution of words
alice = nltk.corpus.gutenberg.words('carroll-alice.txt')
vocab = nltk.FreqDist(alice)

# Select the 1000 most common words
v1000 = [word for (word, _) in vocab.most_common(1000)]

# Create a mapping dictionary to replace less common words with 'UNK'
mapping = defaultdict(lambda: 'UNK')
for v in v1000:
    mapping[v] = v

# Transform the original text, replacing less common words with 'UNK'
alice2 = [mapping[v] for v in alice]

# Display the initial 100 transformed words
print(alice2[:100])

# Determine the number of unique tokens after transformation
print(len(set(alice2)))
```

```
['[', 'Alice', "'", 's', 'Adventures', 'in', 'Wonderland', 'by', 'UNK', 'UNK', 'UNK', 'UNK', 'CHAPTER', 'I', '.', 'Down', 'the', 'Rabbit
1001
```

```
from collections import defaultdict
from nltk.corpus import brown
from operator import itemgetter

counts = defaultdict(int)

for (word, tag) in brown.tagged_words(categories='news', tagset='universal'):
    counts[tag] += 1

counts['NOUN']
sorted(counts)
sorted(counts.items(), key=itemgetter(1), reverse=True)
[t for t, c in sorted(counts.items(), key=itemgetter(1), reverse=True)]
```

```
['NOUN',
 'VERB',
 'ADP',
 '.',
 'DET',
 'ADJ',
```

```
        'ADV',
        'CONJ',
        'PRON',
        'PRT',
        'NUM',
        'X']
```

```
pair = ('NP', 8336)
pair[1]
itemgetter(1)(pair)
```

```
    8336
```

```
from collections import defaultdict
import nltk

last_letters = defaultdict(list)
words = nltk.corpus.words.words('en')

for word in words:
    key = word[-2:]
    last_letters[key].append(word)

last_letters['ly']
last_letters['zy']
```

```
     'bronzy',
     'buzzy',
     'Chazy',
     'cozy',
     'crazy',
     'dazy',
     'dizzy',
     'dozy',
     'enfrenzy',
     'fezzy',
     'fizzy',
     'floozy',
     'fozy',
     'franzy',
     'frenzy',
     'friezy',
     'frizzy',
     'frowzy',
     'furzy',
     'fuzzy',
     'gauzy',
     'gazy',
     'glazy',
     'groszy',
     'hazy',
     'heezy',
     'Izzy',
     'jazzy',
     'Jozy',
     'lawzy',
     'lazy',
     'mazy',
     'mizzy',
     'muzzy',
     'nizy',
     'oozy',
     'quartzy',
     'quizzy',
     'refrenzy',
     'ritzy',
     'Shortzy',
     'sizy',
     'sleazy',
     'sneezy',
     'snoozy',
     'squeezy',
     'Suzy',
     'tanzy',
     'tizzy',
```

```
         wneezy ,
        'woozy',
        'wuzzy',
        'yezzy']
```

```
from collections import defaultdict
import nltk

anagrams = defaultdict(list)
words = nltk.corpus.words.words('en')

for word in words:
    key = ''.join(sorted(word))
    anagrams[key].append(word)

anagrams['aeilnrt']
```

```
['entrail', 'latrine', 'ratline', 'reliant', 'retinal', 'trenail']
```

```
import nltk

words = nltk.corpus.words.words('en')
anagrams = nltk.Index((''.join(sorted(w)), w) for w in words)
anagrams['aeilnrt']
```

```
['entrail', 'latrine', 'ratline', 'reliant', 'retinal', 'trenail']
```

```
from collections import defaultdict
import nltk

pos = defaultdict(lambda: defaultdict(int))
brown_news_tagged = brown.tagged_words(categories='news', tagset='universal')

for ((w1, t1), (w2, t2)) in nltk.bigrams(brown_news_tagged):
    pos[(t1, w2)][t2] += 1

pos[('DET', 'right')]
```

```
defaultdict(int, {'NOUN': 5, 'ADJ': 11})
```

```
from collections import defaultdict
import nltk

counts = defaultdict(int)
for word in nltk.corpus.gutenberg.words('milton-paradise.txt'):
    counts[word] += 1

[word for (word, count) in counts.items() if count == 32]
# Output: ['brought', 'Him', 'virtue', 'Against', 'There', 'thine', 'King', 'mortal', 'every', 'been']
```

```
['mortal',
 'Against',
 'Him',
 'There',
 'brought',
 'King',
 'virtue',
 'every',
 'been',
 'thine']
```

```
pos = {'colorless': 'ADJ', 'ideas': 'N', 'sleep': 'V', 'furiously': 'ADV'}
pos2 = dict((value, key) for (key, value) in pos.items())
pos2['N']
# Output: 'ideas'
```

```
'ideas'
```

```
from collections import defaultdict

pos = {'colorless': 'ADJ', 'ideas': 'N', 'sleep': 'V', 'furiously': 'ADV'}
pos.update({'cats': 'N', 'scratch': 'V', 'peacefully': 'ADV', 'old': 'ADJ'})

pos2 = defaultdict(list)
for key, value in pos.items():
    pos2[value].append(key)

pos2['ADV']
```

```
['furiously', 'peacefully']
```

```
import nltk

pos = {'colorless': 'ADJ', 'ideas': 'N', 'sleep': 'V', 'furiously': 'ADV'}
pos.update({'cats': 'N', 'scratch': 'V', 'peacefully': 'ADV', 'old': 'ADJ'})

pos2 = nltk.Index((value, key) for (key, value) in pos.items())
pos2['ADV']
# Output: ['peacefully', 'furiously']
```

```
['furiously', 'peacefully']
```

```
from nltk.corpus import brown

# Fetching tagged sentences from the 'news' category in the Brown corpus
brown_tagged_sents = brown.tagged_sents(categories='news')

# Fetching untagged sentences from the 'news' category in the Brown corpus
brown_sents = brown.sents(categories='news')



tags = [tag for (word, tag) in brown.tagged_words(categories='news')]
nltk.FreqDist(tags).max()
```

```
'NN'
```

```
raw = 'I do not like green eggs and ham, I do not like them Sam I am!'
tokens = nltk.word_tokenize(raw)
default_tagger = nltk.DefaultTagger('NN')
default_tagger.tag(tokens)
```

```
[('I', 'NN'),
 ('do', 'NN'),
 ('not', 'NN'),
 ('like', 'NN'),
 ('green', 'NN'),
 ('eggs', 'NN'),
 ('and', 'NN'),
 ('ham', 'NN'),
 (',', 'NN'),
 ('I', 'NN'),
 ('do', 'NN'),
 ('not', 'NN'),
 ('like', 'NN'),
 ('them', 'NN'),
 ('Sam', 'NN'),
 ('I', 'NN'),
 ('am', 'NN'),
 ('!', 'NN')]
```

```
default_tagger.evaluate(brown_tagged_sents)
```

```
0.13089484257215028
```

```
patterns = [
...     (r'.*ing$', 'VBG'),               # gerunds
...     (r'.*ed$', 'VBD'),                # simple past
...     (r'.*es$', 'VBZ'),                # 3rd singular present
...     (r'.*ould$', 'MD'),               # modals
...     (r'.*\'s$', 'NN$'),               # possessive nouns
...     (r'.*s$', 'NNS'),                 # plural nouns
...     (r'^-?[0-9]+(\.[0-9]+)?$', 'CD'),  # cardinal numbers
...     (r'.*', 'NN')                     # nouns (default)
... ]


regexp_tagger = nltk.RegexpTagger(patterns)

regexp_tagger.tag(brown_sents[3])
```

```
[('``', 'NN'),
 ('Only', 'NN'),
 ('a', 'NN'),
 ('relative', 'NN'),
 ('handful', 'NN'),
 ('of', 'NN'),
 ('such', 'NN'),
 ('reports', 'NNS'),
 ('was', 'NNS'),
 ('received', 'VBD'),
 ("''", 'NN'),
 (',', 'NN'),
 ('the', 'NN'),
 ('jury', 'NN'),
 ('said', 'NN'),
 (',', 'NN'),
 ('``', 'NN'),
 ('considering', 'VBG'),
 ('the', 'NN'),
 ('widespread', 'NN'),
 ('interest', 'NN'),
 ('in', 'NN'),
 ('the', 'NN'),
 ('election', 'NN'),
 (',', 'NN'),
 ('the', 'NN'),
 ('number', 'NN'),
 ('of', 'NN'),
 ('voters', 'NNS'),
 ('and', 'NN'),
 ('the', 'NN'),
 ('size', 'NN'),
 ('of', 'NN'),
 ('this', 'NNS'),
 ('city', 'NN'),
 ("''", 'NN'),
 ('.', 'NN')]
```

```
regexp_tagger.evaluate(brown_tagged_sents)
```

```
0.20186168625812995
```

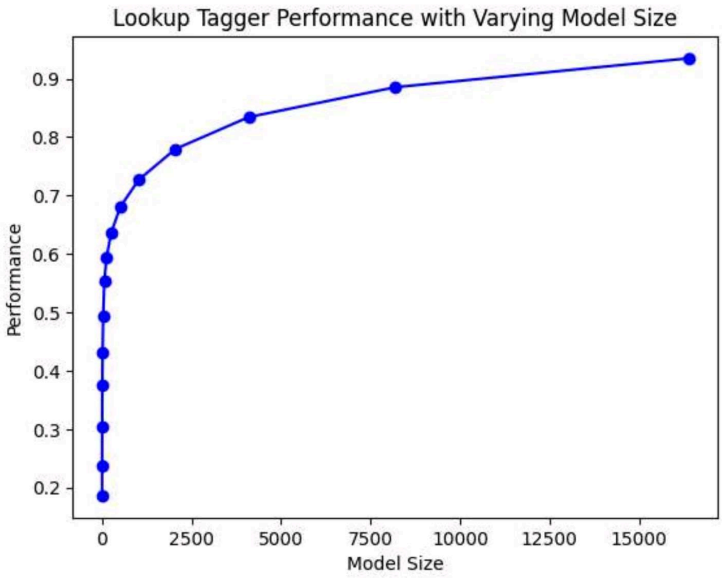
```
import nltk

fd = nltk.FreqDist(brown.words(categories='news'))
cfd = nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))
most_freq_words = fd.most_common(100)
likely_tags = dict((word, cfd[word].max()) for (word, _) in most_freq_words)
baseline_tagger = nltk.UnigramTagger(model=likely_tags)
baseline_tagger.evaluate(brown_tagged_sents)
```

```
0.45578495136941344
```

```
import nltk

sent = brown.sents(categories='news')[3]
baseline_tagger.tag(sent)
```

```
[('``', '``'),
 ('Only', None),
 ('a', 'AT'),
 ('relative', None),
 ('handful', None),
 ('of', 'IN'),
 ('such', None),
 ('reports', None),
 ('was', 'BEDZ'),
 ('received', None),
 ("''", "''"),
 (',', ','),
 ('the', 'AT'),
 ('jury', None),
 ('said', 'VBD'),
 (',', ','),
 ('``', '``'),
 ('considering', None),
 ('the', 'AT'),
 ('widespread', None),
 ('interest', None),
 ('in', 'IN'),
 ('the', 'AT'),
 ('election', None),
 (',', ','),
 ('the', 'AT'),
 ('number', None),
 ('of', 'IN'),
 ('voters', None),
 ('and', 'CC'),
 ('the', 'AT'),
 ('size', None),
 ('of', 'IN'),
 ('this', 'DT'),
 ('city', None),
 ("''", "''"),
 ('.', '.')]
```

```
import nltk

baseline_tagger = nltk.UnigramTagger(model=likely_tags, backoff=nltk.DefaultTagger('NN'))



def performance(cfd, wordlist):
    lt = dict((word, cfd[word].max()) for word in wordlist)
    baseline_tagger = nltk.UnigramTagger(model=lt, backoff=nltk.DefaultTagger('NN'))
    return baseline_tagger.evaluate(brown.tagged_sents(categories='news'))

def display():
    import pylab
    word_freqs = nltk.FreqDist(brown.words(categories='news')).most_common()
    words_by_freq = [w for (w, _) in word_freqs]
    cfd = nltk.ConditionalFreqDist(brown.tagged_words(categories='news'))
    sizes = 2 ** pylab.arange(15)
    perfs = [performance(cfd, words_by_freq[:size]) for size in sizes]
    pylab.plot(sizes, perfs, '-bo')
    pylab.title('Lookup Tagger Performance with Varying Model Size')
    pylab.xlabel('Model Size')
    pylab.ylabel('Performance')
    pylab.show()

display()
```




```
from nltk.corpus import brown

brown_tagged_sents = brown.tagged_sents(categories='news')
brown_sents = brown.sents(categories='news')
unigram_tagger = nltk.UnigramTagger(brown_tagged_sents)
unigram_tagger.tag(brown_sents[2007])
```

```
[('Various', 'JJ'),
 ('of', 'IN'),
 ('the', 'AT'),
 ('apartments', 'NNS'),
 ('are', 'BER'),
 ('of', 'IN'),
 ('the', 'AT'),
 ('terrace', 'NN'),
 ('type', 'NN'),
 (',', ','),
 ('being', 'BEG'),
 ('on', 'IN'),
 ('the', 'AT'),
 ('ground', 'NN'),
 ('floor', 'NN'),
 ('so', 'QL'),
 ('that', 'CS'),
 ('entrance', 'NN'),
 ('is', 'BEZ'),
 ('direct', 'JJ'),
 ('.', '.')]
```

```
unigram_tagger.evaluate(brown_tagged_sents)
```

```
0.9349006503968017
```

```
size = int(len(brown_tagged_sents) * 0.9)
train_sents = brown_tagged_sents[:size]
test_sents = brown_tagged_sents[size:]
unigram_tagger = nltk.UnigramTagger(train_sents)
unigram_tagger.evaluate(test_sents)
```

```
0.8121200039868434
```

```
bigram_tagger = nltk.BigramTagger(train_sents)
bigram_tagger.tag(brown_sents[2007])
```

```
[('Various', 'JJ'),
 ('of', 'IN'),
 ('the', 'AT'),
 ('apartments', 'NNS'),
```

```
        ('are', 'BER'),
        ('of', 'IN'),
        ('the', 'AT'),
        ('terrace', 'NN'),
        ('type', 'NN'),
        (',', ','),
        ('being', 'BEG'),
        ('on', 'IN'),
        ('the', 'AT'),
        ('ground', 'NN'),
        ('floor', 'NN'),
        ('so', 'CS'),
        ('that', 'CS'),
        ('entrance', 'NN'),
        ('is', 'BEZ'),
        ('direct', 'JJ'),
        ('.', '.')]
```

```
unseen_sent = brown_sents[4203]
bigram_tagger.tag(unseen_sent)
```

```
     [('The', 'AT'),
      ('population', 'NN'),
      ('of', 'IN'),
      ('the', 'AT'),
      ('Congo', 'NP'),
      ('is', 'BEZ'),
      ('13.5', None),
      ('million', None),
      (',', None),
      ('divided', None),
      ('into', None),
      ('at', None),
      ('least', None),
      ('seven', None),
      ('major', None),
      ('``', None),
      ('culture', None),
      ('clusters', None),
      ("''", None),
      ('and', None),
      ('innumerable', None),
      ('tribes', None),
      ('speaking', None),
      ('400', None),
      ('separate', None),
      ('dialects', None),
      ('.', None)]
```

```
bigram_tagger.evaluate(test_sents)
```

```
     0.10206319146815508
```

```
t0 = nltk.DefaultTagger('NN')
t1 = nltk.UnigramTagger(train_sents, backoff=t0)
t2 = nltk.BigramTagger(train_sents, backoff=t1)
t2.evaluate(test_sents)
```

```
     0.8452108043456593
```

```
from pickle import dump

output = open('t2.pkl', 'wb')
dump(t2, output, -1)
output.close()
```

```
from pickle import load

input_file = open('t2.pkl', 'rb')
tagger = load(input_file)
input_file.close()
```

```
text = """The board's action shows what free enterprise
...     is up against in our complex maze of regulatory laws ."""
tokens = text.split()
tagger.tag(tokens)
```

```
[('The', 'AT'),
 ("board's", 'NN$'),
 ('action', 'NN'),
 ('shows', 'NNS'),
 ('what', 'WDT'),
 ('free', 'JJ'),
 ('enterprise', 'NN'),
 ('is', 'BEZ'),
 ('up', 'RP'),
 ('against', 'IN'),
 ('in', 'IN'),
 ('our', 'PP$'),
 ('complex', 'JJ'),
 ('maze', 'NN'),
 ('of', 'IN'),
 ('regulatory', 'NN'),
 ('laws', 'NNS'),
 ('.', '.')]
```

```
cfd = nltk.ConditionalFreqDist(
...            ((x[1], y[1], z[0]), z[1])
...            for sent in brown_tagged_sents
...            for x, y, z in nltk.trigrams(sent))
ambiguous_contexts = [c for c in cfd.conditions() if len(cfd[c]) > 1]
sum(cfd[c].N() for c in ambiguous_contexts) / cfd.N()
```

```
0.049297702068029296
```

```
test_tags = [tag for sent in brown.sents(categories='editorial')
...                  for (word, tag) in t2.tag(sent)]
gold_tags = [tag for (word, tag) in brown.tagged_words(categories='editorial')]
print(nltk.ConfusionMatrix(gold_tags, test_tags))
```

```
            |
            |
            |
            |
            |                                                                                         A
            |                                                 -                                       B                   A    A
            |                             *         ,         -         .         :    :              N                   P    P
            |                             -         -         -         -         -    -    A    A    -    A         A    -    -
            |         '                   H         H    -    H         H         H    T    B    B    T    B    A    P    H    T    A
            |    '    '    (    )    *    L    ,    L    -    L    .    L    :    L    L    L    N    L    X    P    $    L    L    T
-----------+-----------------------------------------------------------------------------------------------------------------------
          ' |  <19>   .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
         '' |    . <382>   .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
          ( |    .    .  <95>   .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
          ) |    .    .    .  <95>   .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
          * |    .    .    .    . <304>   .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
       *-HL |    .    .    .    .    3   <.>   .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
          , |    .    .    .    .    .    .<2735>   6    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
       ,-HL |    .    .    .    .    .    .   22   <3>   .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
         -- |    .    .    .    .    .    .    .    . <191>   .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
      ---HL |    .    .    .    .    .    .    .    .    1   <.>   .    .    .    .    .    .    .    .    .    .    .    .    .    .
          . |    .    .    .    .    .    .    .    .    .    .<2976>   4    8    .    .    .    .    .    .    .    .    .    .    .
       .-HL |    .    .    .    .    .    .    .    .    .    .   20   <1>   .    .    .    .    .    .    .    .    .    .    .    .
          : |    .    .    .    .    .    .    .    .    .    .    .    . <108>   .    .    .    .    .    .    .    .    .    .    .
       :-HL |    .    .    .    .    .    .    .    .    .    .    .    .   36   <.>   .    .    .    .    .    .    .    .    .    .
       :-TL |    .    .    .    .    .    .    .    .    .    .    .    .    1    .   <.>   .    .    .    .    .    .    .    .    .
        ABL |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .   <2>   .    .    .    .    .    .    .    .
        ABN |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    . <150>   .    .    .    .    .    .    .
     ABN-TL |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    2   <.>   .    .    .    .    .    .
        ABX |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .  <30>   .    .    .    .    .
         AP |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    . <533>   .    .    .    .
        AP$ |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .   <1>   .    .    .
      AP-HL |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    4    .   <.>   .    .
      AP-TL |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    1    .    .   <.>   .
         AT |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .<5299>
      AT-HL |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .   93
      AT-TL |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .   30
         BE |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
      BE-HL |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
        BED |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
       BEDZ |    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .    .
```

```
            BEDZ* |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
              BEG |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
              BEM |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
              BEN |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
              BER |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
             BER* |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
           BER-HL |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
           BER-TL |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
              BEZ |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
             BEZ* |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
           BEZ-HL |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
           BEZ-TL |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
               CC |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
            CC-HL |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
            CC-TL |   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .   .
               CD |
```

```
from nltk.tbl import demo as brill_demo
brill_demo.demo()
```

```
     Loading tagged data from treebank...
     Read testing data (200 sents/5251 wds)
     Read training data (800 sents/19933 wds)
     Read baseline data (800 sents/19933 wds) [reused the training set]
     Trained baseline tagger
         Accuracy on test set: 0.8366
     Training tbl tagger...
     TBL train (fast) (seqs: 800; tokens: 19933; tpls: 24; min score: 3; min acc: None)
     Finding initial useful rules...
         Found 12799 useful rules.

                B      |
            S   F   r   O  |        Score = Fixed - Broken
            c   i   o   t  |  R     Fixed = num tags changed incorrect -> correct
            o   x   k   h  |  u     Broken = num tags changed correct -> incorrect
            r   e   e   e  |  l     Other = num tags changed incorrect -> incorrect
            e   d   n   r  |  e
```

LAB - 6

1. Exploring Treebank Tagged Corpus

```
#Importing libraries
import nltk, re, pprint
import numpy as np
import pandas as pd
import requests
import matplotlib.pyplot as plt
import seaborn as sns
import pprint, time
import random
from sklearn.model_selection import train_test_split
from nltk.tokenize import word_tokenize
```

```
# reading the Treebank tagged sentences
wsj = list(nltk.corpus.treebank.tagged_sents(tagset='universal'))
```

```
# first few tagged sentences
print(wsj[:40])
```

```
[[('Pierre', 'NOUN'), ('Vinken', 'NOUN'), (',', '.'), ('61', 'NUM'), ('years', 'NOUN'), ('old', 'ADJ'), (',', '.'), ('will', 'VERB'), ('
```

```
# Splitting into train and test
random.seed(1234)
train_set, test_set = train_test_split(wsj,test_size=0.05)

print(len(train_set))
print(len(test_set))
print(train_set[:40])
```

```
3718
196
[[('``', '.'), ('If', 'ADP'), ('you', 'PRON'), ('were', 'VERB'), ('a', 'DET'), ('short-term', 'ADJ'), ('investor', 'NOUN'), (',', '.'),
```

```
# Getting list of tagged words
train_tagged_words = [tup for sent in train_set for tup in sent]
len(train_tagged_words)
```

```
95409
```

```
# tokens
tokens = [pair[0] for pair in train_tagged_words]
tokens[:10]
```

```
['``', 'If', 'you', 'were', 'a', 'short-term', 'investor', ',', 'you', 'might']
```

```
# vocabulary
V = set(tokens)
print(len(V))
```

```
12084
```

```
# number of tags
T = set([pair[1] for pair in train_tagged_words])
len(T)
```

```
12
```

```
print(T)
```

```
{'ADJ', 'NOUN', 'DET', '.', 'VERB', 'NUM', 'X', 'ADP', 'ADV', 'CONJ', 'PRON', 'PRT'}
```

2. HMM(Hidden Markov Model) - POS Tagging Algorithm

Emission Probabilities

```
# computing P(w/t) and storing in T x V matrix
t = len(T)
v = len(V)
w_given_t = np.zeros((t, v))


# compute word given tag: Emission Probability
def word_given_tag(word, tag, train_bag = train_tagged_words):
    tag_list = [pair for pair in train_bag if pair[1]==tag]
    count_tag = len(tag_list)
    w_given_tag_list = [pair[0] for pair in tag_list if pair[0]==word]
    count_w_given_tag = len(w_given_tag_list)

    return (count_w_given_tag, count_tag)


# examples

# large
print("\n", "large")
print(word_given_tag('large', 'ADJ'))
print(word_given_tag('large', 'VERB'))
print(word_given_tag('large', 'NOUN'), "\n")

# will
print("\n", "will")
print(word_given_tag('will', 'MD'))
print(word_given_tag('will', 'NOUN'))
print(word_given_tag('will', 'VERB'))

# book
print("\n", "book")
print(word_given_tag('book', 'NOUN'))
print(word_given_tag('book', 'VERB'))
```

```
 large
(28, 6064)
(0, 12847)
(0, 27421)


 will
(0, 0)
(1, 27421)
(265, 12847)

 book
(7, 27421)
(1, 12847)
```

```
word_given_tag('Twitter','NOUN')
```

```
(0, 27421)
```

Transition Probabilities

```
# compute tag given tag: tag2(t2) given tag1 (t1), i.e. Transition Probability

def t2_given_t1(t2, t1, train_bag = train_tagged_words):
    tags = [pair[1] for pair in train_bag]
    count_t1 = len([t for t in tags if t==t1])
    count_t2_t1 = 0
    for index in range(len(tags)-1):
        if tags[index]==t1 and tags[index+1] == t2:
            count_t2_t1 += 1
    return (count_t2_t1, count_t1)
```

```
# examples
print(t2_given_t1(t2='NOUN', t1='ADJ'))
print(t2_given_t1('ADJ', 'NOUN'))
print(t2_given_t1('NOUN', 'DET'))
print(t2_given_t1('NOUN', 'VERB'))
print(t2_given_t1(',', 'NOUN'))
print(t2_given_t1('PRT', 'PRT'))
print(t2_given_t1('VERB', 'NOUN'))
```

```
(4250, 6064)
(328, 27421)
(5281, 8294)
(1419, 12847)
(0, 27421)
(6, 3042)
(4021, 27421)
```

```
#Please note P(tag|start) is same as P(tag|'.')
print(t2_given_t1('DET', '.'))
print(t2_given_t1('VERB', '.'))
print(t2_given_t1('NOUN', '.'))
print(t2_given_t1('NOUN', '.'))
```

```
(1932, 11072)
(969, 11072)
(2472, 11072)
(2472, 11072)
```

```
t2_given_t1('VERB','DET')
```

```
(333, 8294)
```

```
# creating t x t transition matrix of tags
# each column is t2, each row is t1
# thus M(i, j) represents P(tj given ti)

tags_matrix = np.zeros((len(T), len(T)), dtype='float32')
for i, t1 in enumerate(list(T)):
    for j, t2 in enumerate(list(T)):
        tags_matrix[i, j] = t2_given_t1(t2, t1)[0]/t2_given_t1(t2, t1)[1]
```

```
tags_matrix
```

```
array([[6.74472302e-02, 7.00857520e-01, 4.78232186e-03, 6.29947260e-02,
        1.17084431e-02, 2.12730877e-02, 2.06134561e-02, 7.71767795e-02,
        4.94722975e-03, 1.71503965e-02, 4.94722975e-04, 1.05540901e-02],
       [1.19616352e-02, 2.64140636e-01, 1.33109661e-02, 2.40107954e-01,
        1.46639436e-01, 9.55472048e-03, 2.92841252e-02, 1.76507056e-01,
        1.69942752e-02, 4.27409634e-02, 4.63148672e-03, 4.41267639e-02],
       [2.05570295e-01, 6.36725366e-01, 5.54617774e-03, 1.80853624e-02,
        4.01495062e-02, 2.21847109e-02, 4.58162539e-02, 9.64552723e-03,
        1.20569086e-02, 4.82276344e-04, 3.61707248e-03, 1.20569086e-04],
       [4.47073691e-02, 2.23265901e-01, 1.74494222e-01, 9.34790447e-02,
        8.75180662e-02, 7.92991295e-02, 2.70050578e-02, 9.08598229e-02,
        5.24747111e-02, 5.83453774e-02, 6.59320801e-02, 2.52890168e-03],
       [6.32054210e-02, 1.10453799e-01, 1.35907218e-01, 3.47162746e-02,
        1.69845104e-01, 2.28847209e-02, 2.16937810e-01, 9.18502361e-02,
        8.19646642e-02, 5.44874277e-03, 3.58838625e-02, 3.09021566e-02],
       [3.34337354e-02, 3.58132541e-01, 3.01204808e-03, 1.17771082e-01,
        1.80722885e-02, 1.81024089e-01, 2.09337354e-01, 3.55421677e-02,
        3.01204808e-03, 1.32530117e-02, 1.50602404e-03, 2.59036142e-02],
       [1.69708617e-02, 6.19596541e-02, 5.33141196e-02, 1.62343904e-01,
        2.06211969e-01, 2.88184430e-03, 7.50880539e-02, 1.44732624e-01,
        2.59365998e-02, 1.02465581e-02, 5.60358614e-02, 1.84277937e-01],
       [1.07784428e-01, 3.22711706e-01, 3.24422598e-01, 3.86013687e-02,
        8.12660437e-03, 6.15911037e-02, 3.46449949e-02, 1.72155686e-02,
        1.31522669e-02, 7.48503022e-04, 6.95038512e-02, 1.49700604e-03],
       [1.28957018e-01, 3.26557830e-02, 6.79773390e-02, 1.35288239e-01,
        3.45218271e-01, 3.19893360e-02, 2.23258920e-02, 1.18627124e-01,
        8.03065673e-02, 6.99766725e-03, 1.56614464e-02, 1.39953345e-02],
       [1.18848652e-01, 3.50046426e-01, 1.20241411e-01, 3.29619311e-02,
        1.55060351e-01, 4.17827293e-02, 8.82079825e-03, 5.33890426e-02,
        5.47818020e-02, 4.64252545e-04, 5.84958233e-02, 5.10677788e-03],
       [7.28043169e-02, 2.11093992e-01, 8.85978434e-03, 4.08320501e-02,
        4.86132503e-01, 7.70416018e-03, 9.12942961e-02, 2.31124815e-02,
        3.27426828e-02, 5.39291231e-03, 7.70416018e-03, 1.23266568e-02],
       [8.48126262e-02, 2.50493109e-01, 1.02892831e-01, 4.10913862e-02,
```

```
       3.99408281e-01, 5.65417483e-02, 1.34779746e-02, 2.10387912e-02,
       1.05193956e-02, 2.30111764e-03, 1.54503612e-02, 1.97238661e-03]],
      dtype=float32)
```

```
# convert the matrix to a df for better readability
tags_df = pd.DataFrame(tags_matrix, columns = list(T), index=list(T))
```

```
tags_df
```

	ADJ	NOUN	DET	.	VERB	NUM	X	ADP	
ADJ	0.067447	0.700858	0.004782	0.062995	0.011708	0.021273	0.020613	0.077177	0.004
NOUN	0.011962	0.264141	0.013311	0.240108	0.146639	0.009555	0.029284	0.176507	0.016
DET	0.205570	0.636725	0.005546	0.018085	0.040150	0.022185	0.045816	0.009646	0.012
.	0.044707	0.223266	0.174494	0.093479	0.087518	0.079299	0.027005	0.090860	0.052
VERB	0.063205	0.110454	0.135907	0.034716	0.169845	0.022885	0.216938	0.091850	0.081
NUM	0.033434	0.358133	0.003012	0.117771	0.018072	0.181024	0.209337	0.035542	0.003
X	0.016971	0.061960	0.053314	0.162344	0.206212	0.002882	0.075088	0.144733	0.025
ADP	0.107784	0.322712	0.324423	0.038601	0.008127	0.061591	0.034645	0.017216	0.013
ADV	0.128957	0.032656	0.067977	0.135288	0.345218	0.031989	0.022326	0.118627	0.080
CONJ	0.118849	0.350046	0.120241	0.032962	0.155060	0.041783	0.008821	0.053389	0.054
PRON	0.072804	0.211094	0.008860	0.040832	0.486133	0.007704	0.091294	0.023112	0.032
PRT	0.084813	0.250493	0.102893	0.041091	0.399408	0.056542	0.013478	0.021039	0.010

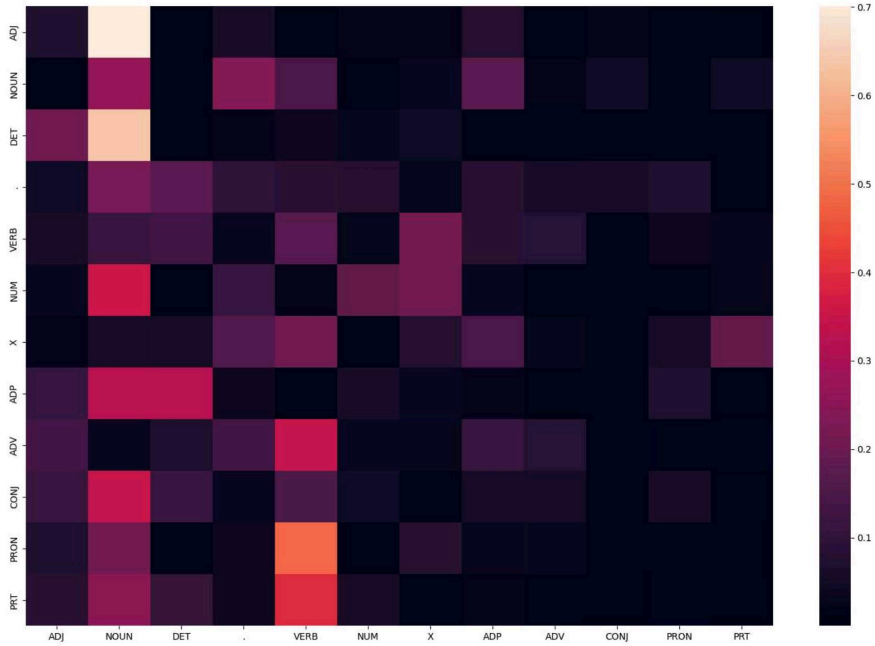
```
tags_df.loc['.', :]
```

```
ADJ     0.044707
NOUN    0.223266
DET     0.174494
.       0.093479
VERB    0.087518
NUM     0.079299
X       0.027005
ADP     0.090860
ADV     0.052475
CONJ    0.058345
PRON    0.065932
PRT     0.002529
Name: ., dtype: float32
```

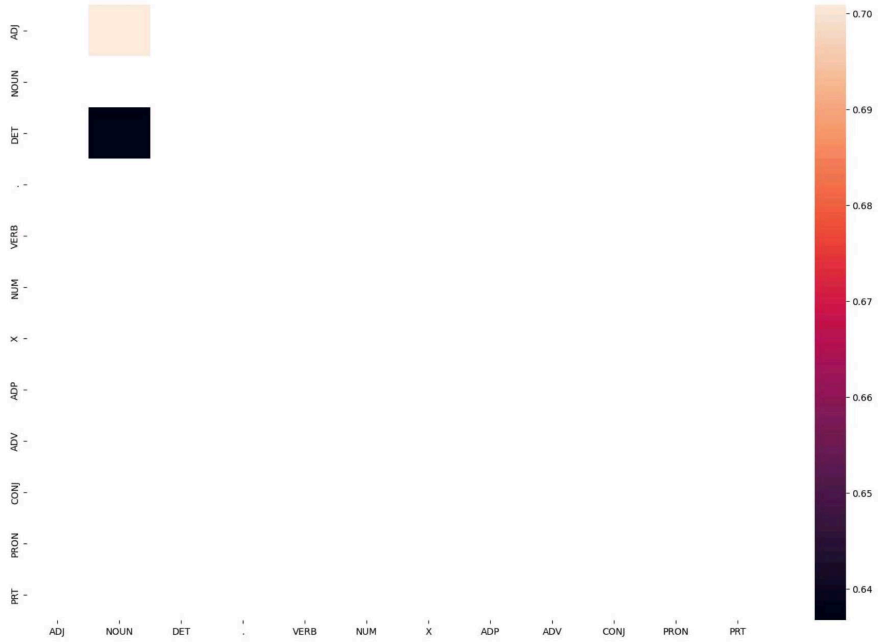
Heatmap of tags matrix

T(i, j) means P(tag j given tag i)

```
plt.figure(figsize=(18, 12))
sns.heatmap(tags_df)
plt.show()
```




```
# frequent tags
# filter the df to get P(t2, t1) > 0.5
tags_frequent = tags_df[tags_df>0.5]
plt.figure(figsize=(18, 12))
sns.heatmap(tags_frequent)
plt.show()
```



3. Viterbi Algorithm

Vanilla Viterbi

```
# Viterbi Heuristic
def Viterbi_vanilla(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            state_probability = emission_p * transition_p
            p.append(state_probability)

        #print('word is {} and list is {}'.format(word,p))
        pmax = max(p)
        # getting state for which probability is maximum
        state_max = T[p.index(pmax)]
        state.append(state_max)
    return list(zip(words, state))
```

Vanilla Laplace smoothened

```
def Viterbi_smoothed(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            emission_p = 0.000001+word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]+0.000001*len(T)
            state_probability = emission_p * transition_p
            p.append(state_probability)

        #print('word is {} and list is {}'.format(word,p))
        pmax = max(p)
        # getting state for which probability is maximum
        state_max = T[p.index(pmax)]
        state.append(state_max)
    return list(zip(words, state))
```

Viterbi with RegExp for unknown words

```
def find_state(word):
    #punc=['"',"'",',','.','(',')','?','[',']',':',';']
    xc=['*']
    if re.search(r'.*(ing|ed|es|ould)$',word.lower()):
        return 'VERB'
    elif re.search(r'to$',str(word).lower()):
        return 'PRT'
    elif re.search(r'^-?[0-9]+(.[0-9]+)?\.*$',str(word).lower()):
        return 'NUM'
    elif '*' in word:
        return 'X'
    elif re.search(r'.*\'s$',word.lower()):
        return 'NOUN'
    elif re.search(r'.*ness$',word.lower()):
        return 'NOUN'
```

```
        elif re.search(r'(The|the|A|a|An|an)$',word):
            return 'ADP'
        elif re.search(r'.*able$',word.lower()):
            return 'ADJ'
        elif re.search(r'.*ly$',word.lower()):
            return 'ADV'
        elif re.search(r'(He|he|She|she|It|it|I|me|Me|You|you|His|his|Her|her|Its|its|my|Your|your|Yours|yours)$',word):
            return 'PRON'
        elif re.search(r'(on|On|at|At|since|Since|For|for|Ago|ago|before|Before|till|Till|until|Until|by|By|Beside|beside|under|Under|below|Below
            return 'ADP'
        elif re.search(r'',word):
            return 'NOUN'
        elif re.search(r'(\'|\"|\.|\(|\)|\?|\[|\]|\:|\;)+',word):
            return '.'
        else:
            return 'NOUN'



    def Viterbi_manual(words, train_bag = train_tagged_words):
        state = []
        T = list(set([pair[1] for pair in train_bag]))

        for key, word in enumerate(words):
            #initialise list of probability column for a given observation
            p = []
            for tag in T:
                if key == 0:
                    transition_p = tags_df.loc['.', tag]
                else:
                    transition_p = tags_df.loc[state[-1], tag]

                # compute emission and state probabilities
                emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
                state_probability = emission_p * transition_p
                p.append(state_probability)

            #print('word is {} and list is {}'.format(word,p))
            pmax = max(p)
            if pmax==0.0:
                state_word=find_state(word)
                state.append(state_word)
                #print(word,':',state_word)
            else:
            # getting state for which probability is maximum
                state_max = T[p.index(pmax)]
                state.append(state_max)
        return list(zip(words, state))
```

Viterbi for unknown words given mostly used tag

```
def Viterbi_common(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            if word in tokens:
                emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            else:
                emission_p=1
            state_probability = emission_p * transition_p
            p.append(state_probability)

        #print('word is {} and list is {}'.format(word,p))
        pmax = max(p)
        if pmax==0.0:
            state_word='NOUN'
            state.append(state_word)
            #print(word,':',state_word)
        else:
        # getting state for which probability is maximum
            state_max = T[p.index(pmax)]
            state.append(state_max)
    return list(zip(words, state))



# Viterbi Heuristic
def Viterbi(words, train_bag = train_tagged_words):
    state = []
    T = list(set([pair[1] for pair in train_bag]))

    for key, word in enumerate(words):
        #initialise list of probability column for a given observation
        p = []
        for tag in T:
            if key == 0:
                transition_p = tags_df.loc['.', tag]
            else:
                transition_p = tags_df.loc[state[-1], tag]

            # compute emission and state probabilities
            if word in tokens:
                emission_p = word_given_tag(words[key], tag)[0]/word_given_tag(words[key], tag)[1]
            else:
                emission_p=1
            state_probability = emission_p * transition_p
            p.append(state_probability)

        #print('word is {} and list is {}'.format(word,p))
        pmax = max(p)
        # getting state for which probability is maximum
        state_max = T[p.index(pmax)]
        state.append(state_max)
    return list(zip(words, state))
```

4. Evaluating on Test Set

```
# Running on entire test dataset would take more than 3-4hrs.
# Let's test our Viterbi algorithm on a few sample sentences of test dataset

#random.seed(1234)

# choose random 5 sents
#rndom = [random.randint(1,len(test_set)) for x in range(10)]

# list of sents
#test_run = [test_set[i] for i in rndom]

# list of tagged words
test_run_base = [tup for sent in test_set for tup in sent]

# list of untagged words
test_tagged_words = [tup[0] for sent in test_set for tup in sent]
test_run_base
```

```
      ('a', 'DET'),
      ('certin', 'NOUN'),
      ('date', 'NOUN'),
      ('at', 'ADP'),
      ('a', 'DET'),
      ('preset', 'ADJ'),
      ('price', 'NOUN'),
      (',', '.'),
      ('known', 'VERB'),
      ('*', 'X'),
      ('as', 'ADP'),
      ('the', 'DET'),
      ('strike', 'NOUN'),
      ('price', 'NOUN'),
      ('.', '.'),
      ('``', '.'),
      ('These', 'DET'),
      ('two', 'NUM'),
      ('exposures', 'NOUN'),
      ('alone', 'ADV'),
      ('represent', 'VERB'),
      ('a', 'DET'),
      ('very', 'ADV'),
      ('substantial', 'ADJ'),
      ('portion', 'NOUN'),
      ('of', 'ADP'),
      ('CS', 'NOUN'),
      ('First', 'NOUN'),
      ('Boston', 'NOUN'),
      ("'s", 'PRT'),
      ('equity', 'NOUN'),
      (',', '.'),
      ("''", '.'),
      ('Moody', 'NOUN'),
      ("'s", 'PRT'),
      ('said', 'VERB'),
      ('*T*-1', 'X'),
      ('.', '.'),
      ('FEDERAL', 'NOUN'),
      ('FUNDS', 'NOUN'),
      (':', '.'),
      ('9', 'NUM'),
      ('1\\/2', 'NUM'),
      ('%', 'NOUN'),
      ('high', 'ADJ'),
      (',', '.'),
      ('8', 'NUM'),
      ('3\\/4', 'NUM'),
      ('%', 'NOUN'),
      ('low', 'ADJ'),
      (',', '.'),
      ('8', 'NUM'),
      ('3\\/4', 'NUM'),
      ('%', 'NOUN'),
      ('near', 'ADP'),
      ('closing', 'ADJ'),
      ('bid', 'NOUN'),
      ...]
```

1. Testing our model on Vanilla Viterbi

```
# tagging the test sentences
start = time.time()
tagged_seq_vanilla = Viterbi_vanilla(test_tagged_words)
check = [i for i, j in zip(tagged_seq_vanilla, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq_vanilla)
end = time.time()
print('Time Taken:',end-start)
print('Accuracy is :',accuracy)
```

```
Time Taken: 550.6328358650208
Accuracy is : 0.9221568255173723
```

```
print('Vanilla Viterbi-Incorrect tagged cases are:')
incorrect_tagged_cases = [[test_run_base[i-1],j] for i, j in enumerate(zip(tagged_seq_vanilla, test_run_base)) if j[0]!=j[1]]
[i[1]for i in incorrect_tagged_cases]
```

```
(('purely', 'ADJ'), ('purely', 'ADV')),
(('foundation', 'ADJ'), ('foundation', 'NOUN')),
(('inferences', 'ADJ'), ('inferences', 'NOUN')),
(('structurally', 'ADJ'), ('structurally', 'ADV')),
(('radically', 'ADJ'), ('radically', 'ADV')),
(('Rail', 'ADJ'), ('Rail', 'NOUN')),
(('enclosed', 'ADJ'), ('enclosed', 'VERB')),
(('transporting', 'ADJ'), ('transporting', 'VERB')),
(('autos', 'ADJ'), ('autos', 'NOUN')),
(('chief', 'NOUN'), ('chief', 'ADJ')),
(('financial-services', 'ADJ'), ('financial-services', 'NOUN')),
(('Private', 'NOUN'), ('Private', 'ADJ')),
(('down', 'ADV'), ('down', 'PRT')),
(('sunlight', 'ADJ'), ('sunlight', 'NOUN')),
(('Possibly', 'ADJ'), ('Possibly', 'ADV')),
(('that', 'ADP'), ('that', 'DET')),
(('more', 'ADJ'), ('more', 'ADV')),
(('clothing', 'ADJ'), ('clothing', 'NOUN')),
(('expense', 'ADJ'), ('expense', 'NOUN')),
(('exceeds', 'ADJ'), ('exceeds', 'VERB')),
(('total', 'ADJ'), ('total', 'NOUN')),
(('a', 'DET'), ('a', 'ADJ')),
(('23,000', 'ADJ'), ('23,000', 'NUM')),
(('accounted', 'ADJ'), ('accounted', 'VERB')),
(('as', 'ADP'), ('as', 'ADV')),
(('sterling', 'ADJ'), ('sterling', 'NOUN')),
(('11.6', 'ADJ'), ('11.6', 'NUM')),
(('solutions', 'ADJ'), ('solutions', 'NOUN')),
(('no', 'DET'), ('no', 'X')),
(('Average', 'NOUN'), ('Average', 'ADJ')),
(('lengthened', 'ADJ'), ('lengthened', 'VERB')),
(('Donoghue', 'ADJ'), ('Donoghue', 'NOUN')),
(('NATIONAL', 'ADJ'), ('NATIONAL', 'NOUN')),
(('ASSOCIATION', 'ADJ'), ('ASSOCIATION', 'NOUN')),
(('9.75', 'ADJ'), ('9.75', 'NUM')),
(('8.70', 'ADJ'), ('8.70', 'NUM')),
(('6\\/2', 'ADJ'), ('6\\/2', 'NUM')),
(('capped', 'VERB'), ('capped', 'ADJ')),
(('tenth', 'ADJ'), ('tenth', 'NOUN')),
(('overstated', 'ADJ'), ('overstated', 'VERB')),
(('Univest', 'ADJ'), ('Univest', 'NOUN')),
(('25.50', 'ADJ'), ('25.50', 'NUM')),
(('467', 'ADJ'), ('467', 'NUM')),
(('Compromises', 'ADJ'), ('Compromises', 'NOUN')),
(('Sidak', 'ADJ'), ('Sidak', 'NOUN')),
(('net', 'NOUN'), ('net', 'ADJ')),
(('versus', 'ADJ'), ('versus', 'CONJ')),
(('5.3', 'ADJ'), ('5.3', 'NUM')),
(('1.61', 'ADJ'), ('1.61', 'NUM')),
(('earlier', 'ADV'), ('earlier', 'ADJ')),
(('that', 'DET'), ('that', 'ADP')),
(('preclude', 'ADJ'), ('preclude', 'VERB')),
(('disseminating', 'ADJ'), ('disseminating', 'VERB')),
(('domestically', 'ADJ'), ('domestically', 'ADV')),
(('before', 'ADP'), ('before', 'ADV')),
(('SWITCHING', 'ADJ'), ('SWITCHING', 'VERB')),
(('TO', 'ADJ'), ('TO', 'PRT')),
(('DEFENSE', 'ADJ'), ('DEFENSE', 'NOUN')),
(('tubes', 'ADJ'), ('tubes', 'NOUN')),
```

2. Testing model on Viterbi Laplace smoothened

```
# tagging the test sentences
start = time.time()
tagged_seq_smooth = Viterbi_smoothed(test_tagged_words)


check = [i for i, j in zip(tagged_seq_smooth, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq_smooth)
end = time.time()
print('Time Taken:',end-start)
print('Accuracy is :',accuracy)
```

```
Time Taken: 543.7705166339874
Accuracy is : 0.9386747674197835
```

```
print('Laplace smoothed Viterbi-Incorrect tagged cases are:')
incorrect_tagged_cases = [[test_run_base[i-1],j] for i, j in enumerate(zip(tagged_seq_smooth, test_run_base)) if j[0]!=j[1]]
[i[1]for i in incorrect_tagged_cases]
```

```
Laplace smoothed Viterbi-Incorrect tagged cases are:
[(('128', 'NOUN'), ('128', 'NUM')),
 (('133', 'NOUN'), ('133', 'NUM')),
 (('135', 'NOUN'), ('135', 'NUM')),
 (('388', 'VERB'), ('388', 'NUM')),
 (('much', 'ADV'), ('much', 'ADJ')),
 (('sympathy', 'VERB'), ('sympathy', 'NOUN')),
 (('in', 'ADP'), ('in', 'PRT')),
 (('American', 'ADJ'), ('American', 'NOUN')),
 (('American', 'ADJ'), ('American', 'NOUN')),
 (('sparking', 'X'), ('sparking', 'VERB')),
 (('posing', 'NOUN'), ('posing', 'VERB')),
 (('as', 'ADP'), ('as', 'ADV')),
 (('Asian', 'ADJ'), ('Asian', 'NOUN')),
 (('stemmed', 'NOUN'), ('stemmed', 'VERB')),
 (('post', 'VERB'), ('post', 'NOUN')),
 (('issue', 'NOUN'), ('issue', 'VERB')),
 (('85.1', 'NOUN'), ('85.1', 'NUM')),
 (('10.5', 'DET'), ('10.5', 'NUM')),
 (('renovated', 'NOUN'), ('renovated', 'VERB')),
 (('hottest', 'NOUN'), ('hottest', 'ADJ')),
 (('chocolate', 'NOUN'), ('chocolate', 'ADJ')),
 (('working', 'NOUN'), ('working', 'ADJ')),
 (('secured', 'NOUN'), ('secured', 'VERB')),
 (('0.99', 'X'), ('0.99', 'NUM')),
 (('319.75', 'VERB'), ('319.75', 'NUM')),
 (('up', 'ADV'), ('up', 'ADP')),
 (('0.60', 'VERB'), ('0.60', 'NUM')),
 (('188.84', 'VERB'), ('188.84', 'NUM')),
 (('get-out-the-vote', 'NOUN'), ('get-out-the-vote', 'ADJ')),
 (('kidnapping', 'DET'), ('kidnapping', 'NOUN')),
 (('language-housekeeper', 'NOUN'), ('language-housekeeper', 'ADJ')),
 (('adapted', 'NOUN'), ('adapted', 'VERB')),
 (('custom-chip', 'NOUN'), ('custom-chip', 'ADJ')),
 (('lagging', 'NOUN'), ('lagging', 'VERB')),
 (('engaging', 'NOUN'), ('engaging', 'ADJ')),
 (('buttoned-down', 'NOUN'), ('buttoned-down', 'ADJ')),
 (('American', 'NOUN'), ('American', 'ADJ')),
 (('lovely', 'DET'), ('lovely', 'ADJ')),
 (('black', 'ADJ'), ('black', 'NOUN')),
 (('white', 'ADJ'), ('white', 'NOUN')),
 (('benign', 'X'), ('benign', 'ADJ')),
 (('drunk', 'NOUN'), ('drunk', 'ADJ')),
 (('away', 'PRT'), ('away', 'ADV')),
 (('out', 'PRT'), ('out', 'ADP')),
 (('limit', 'NOUN'), ('limit', 'VERB')),
 (('lately', 'X'), ('lately', 'ADV')),
 (('as', 'ADP'), ('as', 'ADV')),
 (('entrants', 'VERB'), ('entrants', 'NOUN')),
 (('past', 'ADJ'), ('past', 'NOUN')),
 (('over', 'ADP'), ('over', 'ADJ')),
 (('profit', 'NOUN'), ('profit', 'VERB')),
 (('fleeting', 'X'), ('fleeting', 'ADJ')),
 (('about', 'ADP'), ('about', 'ADV')),
 (('up', 'ADV'), ('up', 'ADP')),
 (('1996', 'NOUN'), ('1996', 'NUM')),
 (('7.63', 'VERB'), ('7.63', 'NUM')),
 (('5.82', 'DET'), ('5.82', 'NUM')),
```

3. Testing model on modified Viterbi with morphological cues (Best model)

```
start = time.time()
tagged_seq_sm = Viterbi_manual(test_tagged_words)
check = [i for i, j in zip(tagged_seq_sm, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq_sm)
end = time.time()
print('Time Taken:',end-start)
print('Accuracy is :',accuracy)
```

```
Time Taken: 380.00050616264343
Accuracy is : 0.9550028479210176
```

```
print('Laplace smoothed Viterbi-Incorrect tagged cases are:')
incorrect_tagged_cases = [[test_run_base[i-1],j] for i, j in enumerate(zip(tagged_seq_sm, test_run_base)) if j[0]!=j[1]]
[i[1]for i in incorrect_tagged_cases]
```

```
Laplace smoothed Viterbi-Incorrect tagged cases are:
[(('much', 'ADV'), ('much', 'ADJ')),
 (('in', 'ADP'), ('in', 'PRT')),
 (('American', 'ADJ'), ('American', 'NOUN')),
 (('American', 'ADJ'), ('American', 'NOUN')),
 (('as', 'ADP'), ('as', 'ADV')),
 (('Asian', 'ADJ'), ('Asian', 'NOUN')),
 (('Indonesia', 'ADP'), ('Indonesia', 'NOUN')),
 (('post', 'VERB'), ('post', 'NOUN')),
 (('BRAMALEA', 'ADP'), ('BRAMALEA', 'NOUN')),
 (('issue', 'NOUN'), ('issue', 'VERB')),
 (('broadcasts', 'VERB'), ('broadcasts', 'NOUN')),
 (('hottest', 'NOUN'), ('hottest', 'ADJ')),
 (('duckling', 'VERB'), ('duckling', 'NOUN')),
 (('consomme', 'PRON'), ('consomme', 'NOUN')),
 (('mignon', 'ADP'), ('mignon', 'NOUN')),
 (('chocolate', 'NOUN'), ('chocolate', 'ADJ')),
 (('radio-station', 'ADP'), ('radio-station', 'NOUN')),
 (('working', 'NOUN'), ('working', 'ADJ')),
 (('up', 'ADV'), ('up', 'ADP')),
 (('get-out-the-vote', 'NOUN'), ('get-out-the-vote', 'ADJ')),
 (('kidnapping', 'VERB'), ('kidnapping', 'NOUN')),
 (('Gates', 'VERB'), ('Gates', 'NOUN')),
 (('language-housekeeper', 'NOUN'), ('language-housekeeper', 'ADJ')),
 (('Gates', 'VERB'), ('Gates', 'NOUN')),
 (('custom-chip', 'NOUN'), ('custom-chip', 'ADJ')),
 (('engaging', 'VERB'), ('engaging', 'ADJ')),
 (('buttoned-down', 'NOUN'), ('buttoned-down', 'ADJ')),
 (('American', 'NOUN'), ('American', 'ADJ')),
 (('lovely', 'ADV'), ('lovely', 'ADJ')),
 (('black', 'ADJ'), ('black', 'NOUN')),
 (('white', 'ADJ'), ('white', 'NOUN')),
 (('benign', 'NOUN'), ('benign', 'ADJ')),
 (('drunk', 'NOUN'), ('drunk', 'ADJ')),
 (('away', 'PRT'), ('away', 'ADV')),
 (('out', 'PRT'), ('out', 'ADP')),
 (('limit', 'NOUN'), ('limit', 'VERB')),
 (('as', 'ADP'), ('as', 'ADV')),
 (('impart', 'NOUN'), ('impart', 'VERB')),
 (('past', 'ADJ'), ('past', 'NOUN')),
 (('over', 'ADP'), ('over', 'ADJ')),
 (('profit', 'NOUN'), ('profit', 'VERB')),
 (('fleeting', 'VERB'), ('fleeting', 'ADJ')),
 (('about', 'ADP'), ('about', 'ADV')),
 (('up', 'ADV'), ('up', 'ADP')),
 (('that', 'ADP'), ('that', 'DET')),
 (('future', 'ADJ'), ('future', 'NOUN')),
 (('put', 'VERB'), ('put', 'NOUN')),
 (('underlying', 'ADJ'), ('underlying', 'VERB')),
 (('taper', 'NOUN'), ('taper', 'VERB')),
 (('that', 'ADP'), ('that', 'DET')),
 (('explore', 'NOUN'), ('explore', 'VERB')),
 (('away', 'ADV'), ('away', 'PRT')),
 (('whipping', 'VERB'), ('whipping', 'ADJ')),
 (('gain', 'NOUN'), ('gain', 'VERB')),
 (('stock-selection', 'ADP'), ('stock-selection', 'ADJ')),
 (('that', 'ADP'), ('that', 'DET')),
 (('coincidental', 'NOUN'), ('coincidental', 'ADJ')),
```

4.Testing model on viterbi with unknown word given mostly used tag

```
start = time.time()
tagged_seq_co = Viterbi_common(test_tagged_words)
check = [i for i, j in zip(tagged_seq_co, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq_co)
end = time.time()
print('Time Taken:',end-start)
print('Accuracy is :',accuracy)
```

```
Time Taken: 365.77018570899963
Accuracy is : 0.9401936586292007
```

```
print('Laplace smoothed Viterbi-Incorrect tagged cases are:')
incorrect_tagged_cases = [[test_run_base[i-1],j] for i, j in enumerate(zip(tagged_seq_co, test_run_base)) if j[0]!=j[1]]
[i[1]for i in incorrect_tagged_cases]
```

```
Laplace smoothed Viterbi-Incorrect tagged cases are:
[(('128', 'NOUN'), ('128', 'NUM')),
 (('133', 'NOUN'), ('133', 'NUM')),
 (('135', 'NOUN'), ('135', 'NUM')),
 (('388', 'VERB'), ('388', 'NUM')),
 (('much', 'ADV'), ('much', 'ADJ')),
 (('sympathy', 'VERB'), ('sympathy', 'NOUN')),
 (('in', 'ADP'), ('in', 'PRT')),
 (('American', 'ADJ'), ('American', 'NOUN')),
 (('American', 'ADJ'), ('American', 'NOUN')),
 (('sparking', 'X'), ('sparking', 'VERB')),
 (('posing', 'NOUN'), ('posing', 'VERB')),
 (('as', 'ADP'), ('as', 'ADV')),
 (('Asian', 'ADJ'), ('Asian', 'NOUN')),
 (('stemmed', 'NOUN'), ('stemmed', 'VERB')),
 (('post', 'VERB'), ('post', 'NOUN')),
 (('issue', 'NOUN'), ('issue', 'VERB')),
 (('85.1', 'NOUN'), ('85.1', 'NUM')),
 (('10.5', 'DET'), ('10.5', 'NUM')),
 (('broadcasts', 'VERB'), ('broadcasts', 'NOUN')),
 (('renovated', 'NOUN'), ('renovated', 'VERB')),
 (('hottest', 'NOUN'), ('hottest', 'ADJ')),
 (('chocolate', 'NOUN'), ('chocolate', 'ADJ')),
 (('working', 'NOUN'), ('working', 'ADJ')),
 (('lenders', 'X'), ('lenders', 'NOUN')),
 (('0.99', 'X'), ('0.99', 'NUM')),
 (('319.75', 'VERB'), ('319.75', 'NUM')),
 (('up', 'ADV'), ('up', 'ADP')),
 (('0.60', 'VERB'), ('0.60', 'NUM')),
 (('188.84', 'VERB'), ('188.84', 'NUM')),
 (('get-out-the-vote', 'NOUN'), ('get-out-the-vote', 'ADJ')),
 (('kidnapping', 'DET'), ('kidnapping', 'NOUN')),
 (('language-housekeeper', 'NOUN'), ('language-housekeeper', 'ADJ')),
 (('adapted', 'NOUN'), ('adapted', 'VERB')),
 (('custom-chip', 'NOUN'), ('custom-chip', 'ADJ')),
 (('lagging', 'NOUN'), ('lagging', 'VERB')),
 (('engaging', 'VERB'), ('engaging', 'ADJ')),
 (('buttoned-down', 'NOUN'), ('buttoned-down', 'ADJ')),
 (('American', 'NOUN'), ('American', 'ADJ')),
 (('lovely', 'DET'), ('lovely', 'ADJ')),
 (('black', 'ADJ'), ('black', 'NOUN')),
 (('white', 'ADJ'), ('white', 'NOUN')),
 (('benign', 'X'), ('benign', 'ADJ')),
 (('drunk', 'NOUN'), ('drunk', 'ADJ')),
 (('away', 'PRT'), ('away', 'ADV')),
 (('out', 'PRT'), ('out', 'ADP')),
 (('limit', 'NOUN'), ('limit', 'VERB')),
 (('lately', 'X'), ('lately', 'ADV')),
 (('as', 'ADP'), ('as', 'ADV')),
 (('entrants', 'VERB'), ('entrants', 'NOUN')),
 (('past', 'ADJ'), ('past', 'NOUN')),
 (('over', 'ADP'), ('over', 'ADJ')),
 (('profit', 'NOUN'), ('profit', 'VERB')),
 (('fleeting', 'X'), ('fleeting', 'ADJ')),
 (('about', 'ADP'), ('about', 'ADV')),
 (('up', 'ADV'), ('up', 'ADP')),
 (('1996', 'NOUN'), ('1996', 'NUM')),
 (('7.63', 'VERB'), ('7.63', 'NUM')),
```

5. Testing model on Viterbi choosing only transisition probability in case emission probability is zero

```
start = time.time()
tagged_seq_t = Viterbi(test_tagged_words)
check = [i for i, j in zip(tagged_seq_t, test_run_base) if i == j]
accuracy = len(check)/len(tagged_seq_t)
end = time.time()
print('Time Taken:',end-start)
print('Accuracy is :',accuracy)
print('Laplace smoothed Viterbi-Incorrect tagged cases are:')
incorrect_tagged_cases = [[test_run_base[i-1],j] for i, j in enumerate(zip(tagged_seq_t, test_run_base)) if j[0]!=j[1]]
[i[1]for i in incorrect_tagged_cases]
```

```
Time Taken: 337.21010279655457
Accuracy is : 0.9401936586292007
Laplace smoothed Viterbi-Incorrect tagged cases are:
[(('128', 'NOUN'), ('128', 'NUM')),
 (('133', 'NOUN'), ('133', 'NUM')),
 (('135', 'NOUN'), ('135', 'NUM')),
 (('388', 'VERB'), ('388', 'NUM')),
 (('much', 'ADV'), ('much', 'ADJ')),
 (('sympathy', 'VERB'), ('sympathy', 'NOUN')),
 (('in', 'ADP'), ('in', 'PRT')),
 (('American', 'ADJ'), ('American', 'NOUN')),
 (('American', 'ADJ'), ('American', 'NOUN')),
 (('sparking', 'X'), ('sparking', 'VERB')),
 (('posing', 'NOUN'), ('posing', 'VERB')),
 (('as', 'ADP'), ('as', 'ADV')),
 (('Asian', 'ADJ'), ('Asian', 'NOUN')),
 (('stemmed', 'NOUN'), ('stemmed', 'VERB')),
 (('post', 'VERB'), ('post', 'NOUN')),
 (('issue', 'NOUN'), ('issue', 'VERB')),
 (('85.1', 'NOUN'), ('85.1', 'NUM')),
 (('10.5', 'DET'), ('10.5', 'NUM')),
 (('broadcasts', 'VERB'), ('broadcasts', 'NOUN')),
 (('renovated', 'NOUN'), ('renovated', 'VERB')),
 (('hottest', 'NOUN'), ('hottest', 'ADJ')),
 (('chocolate', 'NOUN'), ('chocolate', 'ADJ')),
 (('working', 'NOUN'), ('working', 'ADJ')),
 (('lenders', 'X'), ('lenders', 'NOUN')),
 (('0.99', 'X'), ('0.99', 'NUM')),
 (('319.75', 'VERB'), ('319.75', 'NUM')),
 (('up', 'ADV'), ('up', 'ADP')),
 (('0.60', 'VERB'), ('0.60', 'NUM')),
 (('188.84', 'VERB'), ('188.84', 'NUM')),
 (('get-out-the-vote', 'NOUN'), ('get-out-the-vote', 'ADJ')),
 (('kidnapping', 'DET'), ('kidnapping', 'NOUN')),
 (('language-housekeeper', 'NOUN'), ('language-housekeeper', 'ADJ')),
 (('adapted', 'NOUN'), ('adapted', 'VERB')),
 (('custom-chip', 'NOUN'), ('custom-chip', 'ADJ')),
 (('lagging', 'NOUN'), ('lagging', 'VERB')),
 (('engaging', 'VERB'), ('engaging', 'ADJ')),
 (('buttoned-down', 'NOUN'), ('buttoned-down', 'ADJ')),
 (('American', 'NOUN'), ('American', 'ADJ')),
 (('lovely', 'DET'), ('lovely', 'ADJ')),
 (('black', 'ADJ'), ('black', 'NOUN')),
 (('white', 'ADJ'), ('white', 'NOUN')),
 (('benign', 'X'), ('benign', 'ADJ')),
 (('drunk', 'NOUN'), ('drunk', 'ADJ')),
 (('away', 'PRT'), ('away', 'ADV')),
 (('out', 'PRT'), ('out', 'ADP')),
 (('limit', 'NOUN'), ('limit', 'VERB')),
 (('lately', 'X'), ('lately', 'ADV')),
 (('as', 'ADP'), ('as', 'ADV')),
 (('entrants', 'VERB'), ('entrants', 'NOUN')),
 (('past', 'ADJ'), ('past', 'NOUN')),
 (('over', 'ADP'), ('over', 'ADJ')),
 (('profit', 'NOUN'), ('profit', 'VERB')),
 (('fleeting', 'X'), ('fleeting', 'ADJ')),
 (('about', 'ADP'), ('about', 'ADV')),
 (('up', 'ADV'), ('up', 'ADP')),
```

Finding Incorrect tagged cases in vanilla Viterbi algorigtm which are corrected in other models

```
incorrect_tagged_cases = [j for i, j in enumerate(zip(tagged_seq_vanilla, test_run_base)) if j[0]!=j[1]]
#inc2=[j[2] for i, j in enumerate(zip(test_run_base,tagged_seq_vanilla,tagged_seq_sm)) if (j[0]!=j[1])&(j[2][0]==j[1][0])]
```

```
pd.set_option('display.max_rows', None)
cor_word=pd.Series([i[1][0] for i in incorrect_tagged_cases])
cor_pos=pd.Series([i[1][1] for i in incorrect_tagged_cases])
inco=pd.Series([i[0][1] for i in incorrect_tagged_cases])
m2=[j[2] for i, j in enumerate(zip(test_run_base,tagged_seq_vanilla,tagged_seq_smooth)) if (j[0]!=j[1])&(j[2][0]==j[1][0])]
m2_pos=pd.Series([i[1] for i in m2])
m3=[j[2] for i, j in enumerate(zip(test_run_base,tagged_seq_vanilla,tagged_seq_sm)) if (j[0]!=j[1])&(j[2][0]==j[1][0])]
m3_pos=pd.Series([i[1] for i in m3])
m4=[j[2] for i, j in enumerate(zip(test_run_base,tagged_seq_vanilla,tagged_seq_co)) if (j[0]!=j[1])&(j[2][0]==j[1][0])]
m4_pos=pd.Series([i[1] for i in m4])
m5=[j[2] for i, j in enumerate(zip(test_run_base,tagged_seq_vanilla,tagged_seq_t)) if (j[0]!=j[1])&(j[2][0]==j[1][0])]
m5_pos=pd.Series([i[1] for i in m5])
df=pd.DataFrame(cor_word,columns=['Word'])
df.set_index('Word')
df['Correct_POS']=cor_pos
df['Model1_Viterbi_Vanilla']=inco
df['Model2_Laplace']=m2_pos
df['Model3_Morphological']=m3_pos
df['Model4_MostTagBased']=m4_pos
df['Model5_onlyTransistion']=m5_pos
df.drop_duplicates()
```

	Word	Correct_POS	Model1_Viterbi_Vanilla	Model2_Laplace	Model3_Morpho
0	Pretax	NOUN	ADJ	NOUN	
1	128	NUM	ADJ	NOUN	
2	133	NUM	ADJ	NOUN	
3	135	NUM	ADJ	NOUN	
4	388	NUM	ADJ	VERB	
5	much	ADJ	ADV	ADV	
6	sympathy	NOUN	ADJ	VERB	
7	in	PRT	ADP	ADP	
8	American	NOUN	ADJ	ADJ	
10	Farren	NOUN	ADJ	NOUN	
11	sparking	VERB	ADJ	X	
12	posing	VERB	ADJ	NOUN	
13	Participants	NOUN	ADJ	NOUN	
14	Zealand	NOUN	ADJ	NOUN	
15	as	ADV	ADP	ADP	
16	Asian	NOUN	ADJ	ADJ	
17	Nations	NOUN	ADJ	NOUN	
18	Indonesia	NOUN	ADJ	NOUN	
19	Brunei	NOUN	ADJ	NOUN	
20	stemmed	VERB	ADJ	NOUN	
21	post	NOUN	VERB	VERB	
22	directorship	NOUN	ADJ	NOUN	
23	BRAMALEA	NOUN	ADJ	NOUN	
24	issue	VERB	NOUN	NOUN	
25	85.1	NUM	ADJ	NOUN	
26	10.5	NUM	ADJ	DET	
27	broadcasts	NOUN	VERB	NOUN	
28	moons	NOUN	ADJ	NOUN	
29	renovated	VERB	ADJ	NOUN	
30	Roof	NOUN	ADJ	NOUN	

LAB - 7

Cocke-Kasami-Younger (CYK)

```
import itertools
from tabulate import tabulate
import math
from collections import defaultdict
import numpy as np
from binarytree import Node


# Class for grammar rule
class Rule:
    def __init__(self, left, right, prob = 0):
        self.left = left
        self.right = right
        self.prob = prob


# Class grammer that holds the rule
class Grammar:
    def __init__(self, rules):
        self.grammar = {}
        self.reversed_grammar = {}
        for rule in rules:
            self.grammar[rule.left] = self.grammar.get(rule.left, set()).union(set([rule.right]))
            #self.reversed_grammar[rule.right] = self.reversed_grammar.get(rule.right, set()).union(str(rule.left) + "##" + str(rule.prob))
            if rule.right not in self.reversed_grammar:
                self.reversed_grammar[rule.right] = defaultdict(int)
            self.reversed_grammar[rule.right][str(rule.left) + "##" + str(rule.prob)] += 1


# Custom Class for binary tree. It inherites the 'Node' class of binarytree package. This is useful to visualise a tree data structure
class TreeNode(Node):
    def __init__(self, value, log_prob, left, right, word=None):
        value = value if word is None else f"{value}-{word}"
        super().__init__(value, left, right)

        # Calculate the log_score using left, right tree
        left_score = 0 if self.left is None else self.left.score
        right_score = 0 if self.right is None else self.right.score
        self.score = log_prob + left_score + right_score

        # Calculate probability
        self.prob = math.exp(self.score)

    def __repr__(self):
        return str(round(self.prob, 4))


# Calculates a marginal probability of trees
def marginalize_trees(trees):
    if not trees:
        return 0
    prob = sum([tree.prob for tree in trees])
    return prob
```

```
class CKYParser(Grammar):
    # initialize parser
    def __init__(self, rules):
        super().__init__(rules)

    # cky parser algorithm
    def parse(self, words):
        # initialize table
        table = [[""] * i + [set()] * (len(words) - i) for i in range(len(words))]

        # iterate of the columns
        for j in range(len(words)):
            cell = set()
            for word_tag in self.reversed_grammar.get(words[j], {}):
                w, p = word_tag.split('##')
                cell.add(w)

            # initialize diagonal element
            table[j][j] = cell

            # fill up the rows from bottom to up
            for i in reversed(range(j)):
                cell = set()
                for k in range(i, j):
                    rows = table[i][k]
                    cols = table[k + 1][j]
                    permutations = list(itertools.product(rows, cols))

                    # iterate on each permutation
                    for left, right in permutations:
                        check = f"{left} {right}"
                        if check in self.reversed_grammar.keys():
                            cell = cell.union([k.split('##')[0] for k in self.reversed_grammar[check]])
                # update cell value
                table[i][j] = cell if len(cell)>0 else '\u03A6'
        return table

    # weighted cky parser algorithm
    def weigheted_parse(self, words):
        # initialize table
        table = [[""] * i + [set()] * (len(words) - i) for i in range(len(words))]
        tree_list = list()

        # iterate of the columns
        for j in range(len(words)):
            cell = set()
            for word_tag in self.reversed_grammar.get(words[j], {}):
                w, p = word_tag.split('##')
                p = float(p)
                node = TreeNode(w, np.log(p), left=None, right=None, word=words[j])
                cell.add((w, node))

            # initialize diagonal element
            table[j][j] = cell

            # fill up the rows from bottom to up
            for i in reversed(range(j)):
                cell = set()
                for k in range(i, j):
                    rows = table[i][k]
                    cols = table[k + 1][j]
                    permutations = list(itertools.product(rows, cols))

                    # iterate on each permutation
                    for left, right in permutations:
                        l_const, l_node = left
                        r_const, r_node = right
                        check = f"{l_const} {r_const}"
                        if check in self.reversed_grammar.keys():
                            for k in self.reversed_grammar[check]:
                                w, p = k.split('##')
                                p = float(p)
                                new_node = TreeNode(w, np.log(p), left=l_node, right=r_node)
                                cell.add((w, new_node))
                table[i][j] = cell

        # Only add trees that have the root node as the start symbol(S)
```

```
            for tag, root in table[0][-1]:
                if tag == "S":
                    tree_list.append(root)

            return table, tree_list


    sentence = "British left waffles on Falklands"
    tokens = sentence.split(" ")
    rules = [Rule("S", "NP VP"), Rule("NP", "JJ NP"), Rule("VP", "VP NP"), Rule("VP", "VP PP"), Rule("PP", "P NP"), Rule("NP", "British"), Rule(
    cky_parser = CKYParser(rules)


    table = cky_parser.parse(tokens)
    print("CKY Table:")
    print()
    print(tabulate(table, headers=tokens, showindex="always"))
```

```
    CKY Table:

        British       left          waffles       on     Falklands
    --  ------------  ------------  ------------  -----  -----------
     0  {'NP', 'JJ'}  {'S', 'NP'}   {'S'}         Φ      {'S'}
     1                {'VP', 'NP'}  {'VP', 'S'}   Φ      {'S', 'VP'}
     2                              {'VP', 'NP'}  Φ      {'VP'}
     3                                            {'P'}  {'PP'}
     4                                                   {'NP'}
```

```
    sentence = "astronomers saw stars with ears"
    tokens = sentence.split(" ")
    rules = [Rule("S", "NP VP", 1.0), Rule("PP", "P NP", 1.0), Rule("VP", "V NP", 0.7), Rule("VP", "VP PP", 0.3), Rule("P", "with", 1.0), Rule("
    cky_parser = CKYParser(rules)


    table, parse_trees = cky_parser.weigheted_parse(tokens)

    print("Weighted CKY Table:")
    print()
    print(tabulate(table, headers=tokens, showindex="always"))
    print()

    # best tree
    best_tree = max(parse_trees, key=lambda node: node.score)
    print("Most Probable Parse Tree:")
    print()
    print('Score : ', best_tree.prob)
    print(best_tree)
    print()

    # Marginalize over all trees
    print(f"Probability of Sentence marginalized over the trees: {marginalize_trees(parse_trees)}")
```

```
    Weighted CKY Table:

        astronomers    saw                         stars            with          ears
    --  -------------  --------------------------  ---------------  ------------  --------------------------------
     0  {('NP', 0.4)}  set()                       {('S', 0.0504)}  set()         {('S', 0.0027), ('S', 0.0036)}
     1                 {('NP', 0.04), ('V', 1.0)}  {('VP', 0.126)}  set()         {('VP', 0.0091), ('VP', 0.0068)}
     2                                             {('NP', 0.18)}   set()         {('NP', 0.013)}
     3                                                              {('P', 1.0)}  {('PP', 0.18)}
     4                                                                            {('NP', 0.18)}

    Most Probable Parse Tree:

    Score :  0.0036288000000000015

           _______S______
          /              \
    NP-astronomers        __VP_________
                         /             \
                    V-saw          ____NP_______
                                  /             \
                             NP-stars          ___PP___
                                              /        \
                                         P-with     NP-ears


    Probability of Sentence marginalized over the trees: 0.006350400000000003
```

Inference:

The implemented CYK algorithm successfully parses the input string using dynamic programming and bottom-up parsing. The algorithm populates a parsing table with sets of non-terminal symbols, and each set represents the possible constituents of substrings within the input. The final entry in the parsing table indicates whether the input string can be generated by the given grammar. Additionally, the code constructs parse trees for valid input strings, providing insights into the syntactic structure of the input according to the provided grammar rules. This implementation can be employed for various natural language processing tasks, including syntax checking and syntactic analysis of sentences.

Double-click (or enter) to edit

LAB - 8

Minimum Edit Distance Method

```
def min_edit_distance(str1, str2):
    len_str1 = len(str1) + 1
    len_str2 = len(str2) + 1

    # Create a matrix to store minimum edit distances
    matrix = [[0 for _ in range(len_str2)] for _ in range(len_str1)]

    # Initialize the matrix with distances for the first row and column
    for i in range(len_str1):
        matrix[i][0] = i
    for j in range(len_str2):
        matrix[0][j] = j

    # Fill in the matrix with minimum edit distances
    for i in range(1, len_str1):
        for j in range(1, len_str2):
            cost = 0 if str1[i - 1] == str2[j - 1] else 1
            matrix[i][j] = min(
                matrix[i - 1][j] + 1,  # Deletion
                matrix[i][j - 1] + 1,  # Insertion
                matrix[i - 1][j - 1] + cost  # Substitution
            )

    return matrix[len_str1 - 1][len_str2 - 1]


def correct_spelling(word, candidate_words):
    # Find the word in the candidate list with the minimum edit distance
    min_distance = float('inf')
    corrected_word = ""

    for candidate in candidate_words:
        distance = min_edit_distance(word, candidate)
        if distance < min_distance:
            min_distance = distance
            corrected_word = candidate

    return corrected_word, min_distance


# Example usage:
input_word = "happee"  # Replace this with the word you want to correct
candidate_words = ["happy", "happier", "happiness", "happening", "hippie"]

corrected_word, distance = correct_spelling(input_word, candidate_words)
print(f"Original word: {input_word}")
print(f"Corrected spelling: {corrected_word}")
print(f"Minimum edit distance: {distance}")
```

```
Original word: happee
Corrected spelling: happy
Minimum edit distance: 2
```

Inference:

The implemented Minimum Edit Distance algorithm effectively calculates the minimum number of edit operations needed to transform one string into another. In the context of spelling correction, the algorithm successfully identifies the correct spelling from a list of candidates by selecting the word with the minimum edit distance. This approach provides a foundation for spelling correction in applications such as text editors and spell-check systems, enhancing accuracy and improving overall text quality. The algorithm's versatility and efficiency make it suitable for diverse applications requiring similarity evaluation between strings.