Lab Assignment - 3

In [ ]:

```python
from __future__ import division #To avoid integer division
from operator import itemgetter
###Training Phase###

with open("wsj_training.txt", "r") as myfile:
    tr_str = myfile.read()
tr_li = tr_str.split()
num_words_train = len(tr_li)


train_li_words = ['']
train_li_words*= num_words_train

train_li_tags = ['']
train_li_tags*= num_words_train

noun_reduced_list = ['NN','NNS','NNP','NNPS']
verb_reduced_list = ['VB','VBD','VBG','VBN','VBP','VBZ']
adjec_reduced_list = ['JJ', 'JJR', 'JJS']
adv_reduced_list = ['RB', 'RBR', 'RBS']
pronoun_reduced_list = ['PRP', 'PRP$', 'RP']

for i in range(num_words_train):
    temp_li = tr_li[i].split("/")
    train_li_words[i] = temp_li[0]
    if temp_li[1] in noun_reduced_list:
        train_li_tags[i] = 'N'
    elif temp_li[1] in verb_reduced_list:
        train_li_tags[i] = 'V'
    elif temp_li[1] in adjec_reduced_list:
        train_li_tags[i] = 'ADJ'
    elif temp_li[1] in adv_reduced_list:
        train_li_tags[i] = 'ADV'
    elif temp_li[1] in pronoun_reduced_list:
        train_li_tags[i] = 'PRO'
    else:
        train_li_tags[i] = temp_li[1]

k = sorted(list(set(train_li_tags)))
print (k)
dict2_tag_follow_tag_ = {}
"""Nested dictionary to store the transition probabilities
each tag A is a key of the outer dictionary
the inner dictionary is the corresponding value
The inner dictionary's key is the tag B following A
and the corresponding value is the number of times B follows A
"""

dict2_word_tag = {}
"""Nested dictionary to store the emission probabilities.
Each word W is a key of the outer dictionary
The inner dictionary is the corresponding value
The inner dictionary's key is the tag A of the word W
and the corresponding value is the number of times A is a tag of W
"""
```

In [ ]:

```
dict_word_tag_baseline = {}
#Dictionary with word as key and its most frequent tag as value

for i in range(num_words_train-1):
    outer_key = train_li_tags[i]
    inner_key = train_li_tags[i+1]
    dict2_tag_follow_tag_[outer_key]=dict2_tag_follow_tag_.get(outer_key,{})
    dict2_tag_follow_tag_[outer_key][inner_key] = dict2_tag_follow_tag_[outer_key].get(inne
    dict2_tag_follow_tag_[outer_key][inner_key]+=1

    outer_key = train_li_words[i]
    inner_key = train_li_tags[i]
    dict2_word_tag[outer_key]=dict2_word_tag.get(outer_key,{})
    dict2_word_tag[outer_key][inner_key] = dict2_word_tag[outer_key].get(inner_key,0)
    dict2_word_tag[outer_key][inner_key]+=1


"""The 1st token is indicated by being the 1st word of a senetence, that is the word after
Adjusting for the fact that the first word of the document is not accounted for that way
"""

dict2_tag_follow_tag_['.'] = dict2_tag_follow_tag_.get('.',{})
dict2_tag_follow_tag_['.'][train_li_tags[0]] = dict2_tag_follow_tag_['.'].get(train_li_tags
dict2_tag_follow_tag_['.'][train_li_tags[0]]+=1


print (dict2_tag_follow_tag_['IN'])
print (dict2_word_tag['made'])

last_index = num_words_train-1
```

In [ ]:

```
#Accounting for the last word-tag pair
outer_key = train_li_words[last_index]
inner_key = train_li_tags[last_index]
dict2_word_tag[outer_key]=dict2_word_tag.get(outer_key,{})
dict2_word_tag[outer_key][inner_key] = dict2_word_tag[outer_key].get(inner_key,0)
dict2_word_tag[outer_key][inner_key]+=1


"""Converting counts to probabilities in the two nested dictionaries
& also converting the nested dictionaries to outer dictionary with inner sorted lists
"""
for key in dict2_tag_follow_tag_:
    di = dict2_tag_follow_tag_[key]
    s = sum(di.values())
    for innkey in di:
        di[innkey] /= s
    di = di.items()
    di = sorted(di,key=lambda x: x[0])
    dict2_tag_follow_tag_[key] = di

for key in dict2_word_tag:
    di = dict2_word_tag[key]
    dict_word_tag_baseline[key] = max(di, key=di.get)
    s = sum(di.values())
    for innkey in di:
        di[innkey] /= s
    di = di.items()
    di = sorted(di,key=lambda x: x[0])
    dict2_word_tag[key] = di
```

In [ ]:

```
with open("test.txt", "r") as myfile:
    te_str = myfile.read()

te_li = te_str.split()
num_words_test = len(te_li)

test_li_words = ['']
test_li_words*= num_words_test

test_li_tags = ['']
test_li_tags*= num_words_test

output_li = ['']
output_li*= num_words_test

output_li_baseline = ['']
output_li_baseline*= num_words_test

num_errors = 0
num_errors_baseline = 0
```

In [ ]:

```
for i in range(num_words_test):
    temp_li = te_li[i].split("/")
    test_li_words[i] = temp_li[0]
    if temp_li[1] in noun_reduced_list:
        test_li_tags[i] = 'N'
    elif temp_li[1] in verb_reduced_list:
        test_li_tags[i] = 'V'
    elif temp_li[1] in adjec_reduced_list:
        test_li_tags[i] = 'ADJ'
    elif temp_li[1] in adv_reduced_list:
        test_li_tags[i] = 'ADV'
    elif temp_li[1] in pronoun_reduced_list:
        test_li_tags[i] = 'PRO'
    else:
        test_li_tags[i] = temp_li[1]


    output_li_baseline[i] = dict_word_tag_baseline.get(temp_li[0],'')
    #If unknown word - tag = 'N'
    if output_li_baseline[i]=='':
        output_li_baseline[i]='N'


    if output_li_baseline[i]!=test_li_tags[i]:
        num_errors_baseline+=1


    if i==0:    #Accounting for the 1st word in the test document for the Viterbi
        di_transition_probs = dict2_tag_follow_tag_['.']
    else:
        di_transition_probs = dict2_tag_follow_tag_[output_li[i-1]]

    di_emission_probs = dict2_word_tag.get(test_li_words[i],'')

    #If unknown word  - tag = 'N'
    if di_emission_probs=='':
        output_li[i]='N'

    else:
        max_prod_prob = 0
        counter_trans = 0
        counter_emis =0
        prod_prob = 0
        while counter_trans < len(di_transition_probs) and counter_emis < len(di_emission_p
            tag_tr = di_transition_probs[counter_trans][0]
            tag_em = di_emission_probs[counter_emis][0]
            if tag_tr < tag_em:
                counter_trans+=1
            elif tag_tr > tag_em:
                counter_emis+=1
            else:
                prod_prob = di_transition_probs[counter_trans][1] * di_emission_probs[count
                if prod_prob > max_prod_prob:
                    max_prod_prob = prod_prob
                    output_li[i] = tag_tr
                    print ("i=",i," and output=",output_li[i])
                counter_trans+=1
                counter_emis+=1

    if output_li[i]=='': #In case there are no matching entries between the transition tags
        output_li[i] = max(di_emission_probs,key=itemgetter(1))[0]

    if output_li[i]!=test_li_tags[i]:
        num_errors+=1
```

In [1]:
```
print ("Fraction of errors (Baseline) :",(num_errors_baseline/num_words_test))
print ("Fraction of errors (Viterbi):",(num_errors/num_words_test))

print ("Tags suggested by Baseline Algorithm:", output_li_baseline)

print ("Tags suggested by Viterbi Algorithm:", output_li)

print ("Correct tags:",test_li_tags)
```

```
[')', ',', '.', ':', 'ADJ', 'ADV', 'CC', 'CD', 'DT', 'FW', 'IN', 'MD', 'N', 'POS', 'PRO',
'TO', 'V', 'WDT', 'WP', '``']
{'DT': 19, 'N': 17, 'PRO': 3, 'CD': 7, ',': 1, 'ADV': 1, 'ADJ': 3}
{'V': 2}
i= 1  and output= V
i= 2  and output= ADJ
Fraction of errors (Baseline) : 0.0
Fraction of errors (Viterbi): 0.0
Tags suggested by Baseline Algorithm: ['N', 'V', 'ADJ', 'N']
Tags suggested by Viterbi Algorithm: ['N', 'V', 'ADJ', 'N']
Correct tags: ['N', 'V', 'ADJ', 'N']
```

In [ ]:

In [3]:
```
import pandas as pd
df =  pd.read_csv(r"C:\Users\raval\jupyter_notebook\NLP\words_pos.csv")
df.head()
```

Out[3]:

	Unnamed: 0	word	pos_tag
0	0	aa	NN
1	1	aaa	NN
2	2	aah	NN
3	3	aahed	VBN
4	4	aahing	VBG

In [4]:
```
df = df.drop(["Unnamed: 0"],axis=1)
```

In [13]:
```
tuple(df["word"]),tuple(df["pos_tag"])
```

```
  'aaron',
  'aaronic',
  'aaronical',
  'aaronite',
  'aaronitic',
  'aarrgh',
  'aarrghh',
  'aaru',
  'aas',
  'aasvogel',
  'aasvogels',
  'ab',
  'aba',
  'ababdeh',
  'ababua',
  'abac',
  'abaca',
  'abacay',
  'abacas',
  'abacate',
  'abacaxi',
```

In [16]:

```
# Initialize lists as empty lists
train_li_words = []
train_li_tags = []

# Define reduced tag lists
noun_reduced_list = ['NN', 'NNS', 'NNP', 'NNPS']
verb_reduced_list = ['VB', 'VBD', 'VBG', 'VBN', 'VBP', 'VBZ']
adjec_reduced_list = ['JJ', 'JJR', 'JJS']
adv_reduced_list = ['RB', 'RBR', 'RBS']
pronoun_reduced_list = ['PRP', 'PRP$', 'RP']

# Assuming df is a DataFrame containing word and pos_tag columns
for index, row in df.iterrows():
    train_li_words.append(row["word"])
    if row["pos_tag"] in noun_reduced_list:
        train_li_tags.append('N')
    elif row["pos_tag"] in verb_reduced_list:
        train_li_tags.append('V')
    elif row["pos_tag"] in adjec_reduced_list:
        train_li_tags.append('ADJ')
    elif row["pos_tag"] in adv_reduced_list:
        train_li_tags.append('ADV')
    elif row["pos_tag"] in pronoun_reduced_list:
        train_li_tags.append('PRO')
    else:
        train_li_tags.append(row["pos_tag"])

# Training phase
dict2_tag_follow_tag_ = {}
dict2_word_tag = {}
dict_word_tag_baseline = {}

for i in range(len(train_li_words) - 1):
    outer_key = train_li_tags[i]
    inner_key = train_li_tags[i + 1]
    dict2_tag_follow_tag_.setdefault(outer_key, {}).setdefault(inner_key, 0)
    dict2_tag_follow_tag_[outer_key][inner_key] += 1

    outer_key = train_li_words[i]
    inner_key = train_li_tags[i]
    dict2_word_tag.setdefault(outer_key, {}).setdefault(inner_key, 0)
    dict2_word_tag[outer_key][inner_key] += 1

    dict_word_tag_baseline.setdefault(outer_key, inner_key)

dict2_tag_follow_tag_['.'] = {train_li_tags[0]: 1}  # Accounting for the first word

# Converting counts to probabilities
for key in dict2_tag_follow_tag_:
    di = dict2_tag_follow_tag_[key]
    s = sum(di.values())
    for innkey in di:
        di[innkey] /= s
    di = di.items()
    di = sorted(di, key=lambda x: x[0])
    dict2_tag_follow_tag_[key] = di

for key in dict2_word_tag:
    di = dict2_word_tag[key]
    s = sum(di.values())
    for innkey in di:
        di[innkey] /= s
    di = di.items()
    di = sorted(di, key=lambda x: x[0])
    dict2_word_tag[key] = di

# Testing phase
with open("test.txt", "r") as myfile:
    te_str = myfile.read()

te_li = te_str.split()
num_words_test = len(te_li)
```

```
test_li_words = []
test_li_tags = []
output_li = []
output_li_baseline = []
num_errors = 0
num_errors_baseline = 0

for i in range(num_words_test):
    temp_li = te_li[i].split("/")
    test_li_words.append(temp_li[0])

    if temp_li[1] in noun_reduced_list:
        test_li_tags.append('N')
    elif temp_li[1] in verb_reduced_list:
        test_li_tags.append('V')
    elif temp_li[1] in adjec_reduced_list:
        test_li_tags.append('ADJ')
    elif temp_li[1] in adv_reduced_list:
        test_li_tags.append('ADV')
    elif temp_li[1] in pronoun_reduced_list:
        test_li_tags.append('PRO')
    else:
        test_li_tags.append(temp_li[1])

    output_li_baseline.append(dict_word_tag_baseline.get(temp_li[0], 'N'))

    if output_li_baseline[i] != test_li_tags[i]:
        num_errors_baseline += 1

    if i == 0:
        di_transition_probs = dict2_tag_follow_tag_['.']
    else:
        di_transition_probs = dict2_tag_follow_tag_[output_li[i - 1]]

    di_emission_probs = dict2_word_tag.get(test_li_words[i], '')

    if di_emission_probs == '':
        output_li.append('N')
    else:
        max_prod_prob = 0
        best_tag = 'N'
        for tag_tr, prob_tr in di_transition_probs:
            if tag_tr in di_emission_probs:
                prob_em = di_emission_probs[tag_tr]
                prod_prob = prob_tr * prob_em
                if prod_prob > max_prod_prob:
                    max_prod_prob = prod_prob
                    best_tag = tag_tr
        output_li.append(best_tag)

    if output_li[i] != test_li_tags[i]:
        num_errors += 1

print("Fraction of errors (Baseline):", (num_errors_baseline / num_words_test))
print("Fraction of errors (Viterbi):", (num_errors / num_words_test))

print("Tags suggested by Baseline Algorithm:", output_li_baseline)
print("Tags suggested by Viterbi Algorithm:", output_li)
print("Correct tags:", test_li_tags)
```

```
Fraction of errors (Baseline): 0.0
Fraction of errors (Viterbi): 0.5
Tags suggested by Baseline Algorithm: ['N', 'V', 'ADJ', 'N']
Tags suggested by Viterbi Algorithm: ['N', 'N', 'N', 'N']
Correct tags: ['N', 'V', 'ADJ', 'N']
```

trying to improve viterbi

In [20]:

```
# Testing phase
with open("test.txt", "r") as myfile:
    te_str = myfile.read()

te_li = te_str.split()
num_words_test = len(te_li)

test_li_words = []
test_li_tags = []
output_li = []
output_li_baseline = []
num_errors = 0
num_errors_baseline = 0

# Smoothing parameter for Laplace smoothing
alpha = 0.01

for i in range(num_words_test):
    temp_li = te_li[i].split("/")
    test_word = temp_li[0]

    test_tag = None
    if temp_li[1] in noun_reduced_list:
        test_tag = 'N'
    elif temp_li[1] in verb_reduced_list:
        test_tag = 'V'
    elif temp_li[1] in adjec_reduced_list:
        test_tag = 'ADJ'
    elif temp_li[1] in adv_reduced_list:
        test_tag = 'ADV'
    elif temp_li[1] in pronoun_reduced_list:
        test_tag = 'PRO'
    else:
        test_tag = temp_li[1]

    test_li_words.append(test_word)
    test_li_tags.append(test_tag)

    output_li_baseline.append(dict_word_tag_baseline.get(test_word, 'N'))

    if output_li_baseline[i] != test_tag:
        num_errors_baseline += 1

    if i == 0:
        di_transition_probs = dict2_tag_follow_tag_['.']
    else:
        # Expand context window to consider multiple previous tags
        prev_tags = output_li[max(0, i - 3):i]
        tag_set = set(tag for sublist in [dict2_tag_follow_tag_.get(prev_tag, []) for prev_
        di_transition_probs = {tag: alpha for tag in tag_set}
        for prev_tag in prev_tags:
            if prev_tag in dict2_tag_follow_tag_:
                for next_tag, prob in dict2_tag_follow_tag_[prev_tag]:
                    di_transition_probs[next_tag] += prob

    di_emission_probs = dict2_word_tag.get(test_word, {})

    if not di_emission_probs:  # Unknown word
        output_li.append('N')
        if 'N' != test_tag:  # Increment error count if 'N' is not the correct tag
            num_errors += 1
    else:
        max_prod_prob = 0
        best_tag = 'N'
        for tag_tr, prob_tr in di_transition_probs.items():
            if tag_tr in di_emission_probs:
                prob_em = di_emission_probs[tag_tr]
                prod_prob = prob_tr * prob_em
                if prod_prob > max_prod_prob:
                    max_prod_prob = prod_prob
                    best_tag = tag_tr
        output_li.append(best_tag)
    if output_li[i] != test_tag:
        num_errors += 1

print("Fraction of errors (Baseline):", (num_errors_baseline / num_words_test))
print("Fraction of errors (Viterbi):", (num_errors / num_words_test))

print("Tags suggested by Baseline Algorithm:", output_li_baseline)
print("Tags suggested by Viterbi Algorithm:", output_li)
print("Correct tags:", test_li_tags)
```

```
Fraction of errors (Baseline): 0.0
Fraction of errors (Viterbi): 0.5
Tags suggested by Baseline Algorithm: ['N', 'V', 'ADJ', 'N']
Tags suggested by Viterbi Algorithm: ['N', 'N', 'N', 'N']
Correct tags: ['N', 'V', 'ADJ', 'N']
```

In [ ]: