HW6 Q3

In [1]:
```
import numpy as np
import pickle
import matplotlib.pyplot as plt
import copy

from tabulate import tabulate
import pandas as pd
```

In [2]:
```
with open('alice_spelling.pkl','rb') as f:
    u = pickle._Unpickler(f)
    u.encoding = 'latin1'
    data = u.load()

#Take a look at how the data looks, and let's make some helper functions.
# data = pickle.load(open('alice_spelling.pkl','rb'))
vocab = np.unique(data['corpus'])
V = len(vocab)

## CORRECT VS INCORRECT CORPUS
##For now, we will hold onto both the correct and incorrect corpuses. Later, you will only process the
 incorrect corpus, and the correct corpus is only used as a reference to check for recovery accuracy.
def recovery_rate(new_corpus, correct_corpus):
    wrong = 0
    for k in range(len(new_corpus)):
        if new_corpus[k] != correct_corpus[k]:
            wrong += 1
    return 1.- wrong/(len(new_corpus)+0.)
print('current recovery rate', recovery_rate(data['corpus'],data['corrupted_corpus'] ))

## Probability of a word mispelling
## We will use the following function to predict whether a misspelled word was actually another word.
# To avoid numerical issues, we make sure that the probablity is always something nonzero.
def prob_correct(word1,word2):
    SMALLNUM = 0.0000001
    if len(word1) != len(word2): return SMALLNUM
    num_wrong = np.sum(np.array([word1[k] == word2[k] for k in range(len(word1))]))
    return np.maximum(num_wrong / (len(word1)),SMALLNUM)

# print('prob not misspelling alice vs alace', prob_correct('alice','alace'))
print('prob not misspelling alice vs alace', prob_correct('alice','alace'))
print('prob not misspelling alice vs earth', prob_correct('alice','earth'))
print('prob not misspelling machinelearning vs machinedreaming', prob_correct('machinelearning','machin
edreaming'))
print('prob not misspelling machinelearning vs artificalintell', prob_correct('machinelearning','artifi
calintell'))


##HASHING
#all of our objects should be vectors of length V or matrices which are V x V.
#the kth word in the vocab list is represented by the kth element of the vector, and the relationship b
etween the i,jth words is represented in the i,jth element in the matrix.
# to easily go between the word indices and words themselves, we need to make a hash table.
vocab_hash = {}
for k in range(len(vocab)):
    vocab_hash[vocab[k]] = k

#now, to access the $k$th word, we do vocab[k]. To access the index of a word, we call vocab_hash[wor
d].
```

```
current recovery rate 0.7716434266712013
prob not misspelling alice vs alace 0.8
prob not misspelling alice vs earth 1e-06
prob not misspelling machinelearning vs machinedreaming 0.6666666666666666
prob not misspelling machinelearning vs artificalintell 1e-06
```

In [3]:
```
## FILL ME IN ##

#WORD FREQUENCY
#create an array of length V where V[k] returns the normalized frequency of word k in the entire data c
orpus.
# Do so by filling in this function.
def get_word_prob(corpus):
    wordList,countArray = np.unique(corpus, return_counts=True)
    totalWords = sum(countArray)
    word_prob = np.zeros(len(wordList))
    for i in range(len(wordList)):
        word_prob[i] = countArray[i] / totalWords

    return word_prob

word_prob =  get_word_prob(data['corpus'])

#report the answer of the following:
print('prob. of "alice"', word_prob[vocab_hash['alice']])
print('prob. of "queen"', word_prob[vocab_hash['queen']])
print('prob. of "chapter"', word_prob[vocab_hash['chapter']])

def getPrevWordAndCurrentWordDict():
    prevWordAndCurrentWordDict = {}
    prevWord = data['corpus'][0]
    for i in range(1,len(data['corpus'])):
        word = data['corpus'][i]
        if prevWord not in prevWordAndCurrentWordDict:
            prevWordAndCurrentWordDict[prevWord] = {}
            prevWordAndCurrentWordDict[prevWord][word] = 1
        elif word not in  prevWordAndCurrentWordDict[prevWord]:
            prevWordAndCurrentWordDict[prevWord][word] = 1
        elif word in  prevWordAndCurrentWordDict[prevWord]:
            prevWordAndCurrentWordDict[prevWord][word] += 1
        else:
            print("Shouldn't happen")
        prevWord = word
    return prevWordAndCurrentWordDict

## FILL ME IN ##

# Pr(word | prev word)
# Using the uncorrupted corpus, accumulate the conditional transition probabilities. Do so via this for
mula:
# pr(word | prev) = max(# times 'prev' preceded 'word' , 1) / # times prev appears
# where again, we ensure that this number is never 0 with some small smoothing.
def get_transition_matrix(corpus):

    transition_matrix = np.zeros((len(vocab),len(vocab)))
    wordList,countArray = np.unique(corpus, return_counts=True)
    prevWordAndCurrentWordDict = getPrevWordAndCurrentWordDict()
    for word in range(len(vocab)):
        wordString = wordList[word]
        for prevWord in range(len(vocab)):
            prevWordString = wordList[prevWord]
            prevWordPreceded = 0
            if wordString in prevWordAndCurrentWordDict[prevWordString]:
                prevWordPreceded = prevWordAndCurrentWordDict[prevWordString][wordString]
            occurences = max(prevWordPreceded, 1)
            transition_matrix[word][prevWord] = occurences / countArray[prevWord]
    return transition_matrix
transition_matrix = get_transition_matrix(data['corpus'])

print('prob. of "the alice"', transition_matrix[vocab_hash['alice'],vocab_hash['the']])
print('prob. of "the queen"', transition_matrix[vocab_hash['queen'],vocab_hash['the']])
print('prob. of "the chapter"', transition_matrix[vocab_hash['hatter'],vocab_hash['the']])
```

```
prob. of "alice" 0.014548615047424706
prob. of "queen" 0.002569625514869818
prob. of "chapter" 0.0009069265230699947
prob. of "the alice" 0.0006105006105006105
prob. of "the queen" 0.03968253968253968
prob. of "the chapter" 0.031135531135531136
```

In [4]:
```
#The prior probabilities are just the word frequencies
prior = word_prob

#write a function that returns the emission probability of a potentially misspelled word, by comparing
 its probabilities against every word in the correct vocabulary
def get_emission(mword):
    emission_prob = np.zeros(len(vocab))
    for index, word in enumerate(vocab):
        emission_prob[index] = prob_correct(mword,word)
    return emission_prob

#find the 10 closest words to 'abice' and report them
idx = np.argsort(get_emission('abice'))[::-1]
print([vocab[j] for j in idx[:10]])
```

```
['abide', 'alice', 'above', 'voice', 'alive', 'twice', 'thick', 'dance', 'stick', 'prize']
```

In [5]:
```
#now we reduce our attention to a small segment of the corrupted corpus
corrupt_corpus =   data['corrupted_corpus'][:1000]
correct_corpus =   data['corpus'][:1000]
```

In [6]:
```
def normalize(vector):
    return vector/np.sum(vector)
```

In [7]:
```
# encode the HMM spelling corrector.
# To debug, you can see the first hundred words of both the corrupted and proposed corpus,
# to see if spelling words got corrupted.
# report the recovery rate of the proposed (corrected) corpus.

totalStates = len(transition_matrix)
node_values_fwd = np.zeros((len(corrupt_corpus), totalStates))
for i, sequence_val in enumerate(correct_corpus):
    if (i == 0):
        word_prob =  get_word_prob(data['corpus'])
        start_probs =  word_prob[vocab_hash[sequence_val]]
        emission = get_emission(sequence_val)
        firstStateBeforeNormalisation = start_probs * emission
        node_values_fwd[i, :] = normalize(firstStateBeforeNormalisation)
    else:
        emission = get_emission(sequence_val)
        nextStateBeforeNormalization = np.multiply(emission,np.dot(transition_matrix ,node_values_fwd[i
-1, :]))
        node_values_fwd[i, :] = normalize(nextStateBeforeNormalization)

totalStates = len(transition_matrix)
node_values_bwd = np.zeros((len(corrupt_corpus), totalStates))
for i, e in reversed(list(enumerate(corrupt_corpus))):
    if (i == len(corrupt_corpus)-1):
        word_prob =  get_word_prob(data['corpus'])
        start_probs =  word_prob[vocab_hash[sequence_val]]
        emission = get_emission(sequence_val)
        firstStateBeforeNormalisation = start_probs * emission
        node_values_bwd[i, :] = normalize(firstStateBeforeNormalisation)
    else:
        emission = get_emission(sequence_val)
        nextStateBeforeNormalization = np.multiply(emission,np.dot(transition_matrix ,node_values_bwd[i
+1, :]))
        node_values_bwd[i, :] = normalize(nextStateBeforeNormalization)
```

In [8]:
```
forward_backward = np.multiply(node_values_fwd, node_values_bwd)
row_index = np.argmax(forward_backward, axis=1)
proposed_corpus = []
for index in row_index:
    proposed_corpus.append(vocab[index])

results = []
results.append(('correct_corpus','corrupt_corpus',recovery_rate(corrupt_corpus, correct_corpus)))
results.append(('correct_corpus','proposed_corpus',recovery_rate(proposed_corpus, correct_corpus)))
columns = ['corpus_A','corpus_B', 'recovery_rate']
df = pd.DataFrame(results, columns=columns)
print(tabulate(df, headers='keys', tablefmt='psql'))
```

```
+----+----------------+----------------+-----------------+
|    | corpus_A       | corpus_B       |   recovery_rate |
|----+----------------+----------------+-----------------|
|  0 | correct_corpus | corrupt_corpus |           0.759 |
|  1 | correct_corpus | proposed_corpus |           0.804 |
+----+----------------+----------------+-----------------+
```