Sentiment Analysis System using HMM

Hidden Markov Model (HMM) is a statistical Markov model in which the system being modeled is assumed to be a Markov process with unobserved (i.e. hidden) states.

In class, we have learnt the definition of HMM and we will be using it in this system to estimate our emission and transmission parameters.

In this project, x's are the natural language words, and y's are the tags (such as O, B-positive).

Import relevant libraries

NumPy is the fundamental package for scientific computing with Python.

pandas is an open source, BSD-licensed library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming. We will be using it to handle our data.

In [1]:

```
import numpy as np
import pandas as pd
```

Process File Functions

We write a function to process the file that we are given to a suitable format for data handling.

In [39]:

```
def get_data(filename):
    f = open(filename,'r', encoding="utf8")
    lines = f.readlines()
    datas = []

    start = 0
    for i in range(len(lines)):
        if lines[i] == '\n':
            datas.append(lines[start:i])
            start = i+1
        lines[i] = lines[i].replace('\n','')
        lines[i] = tuple(lines[i].split(' '))

    # check formatting
    for i in range(len(datas)):
        for j in range(len(datas[i])):
            assert len(datas[i][j])==2


    for i in range(len(datas)):
        data = datas[i]
        x = [word[0] for word in data]
        y = [word[1] for word in data]
        datas[i] = [x,y]
```

```
    all_x = []
    for i in range(len(datas)):
        for j in range(len(datas[i][0])):
            all_x.append(datas[i][0][j])
    x_set = frozenset(all_x)

    all_y = []
    for i in range(len(datas)):
        for j in range(len(datas[i][0])):
            all_y.append(datas[i][1][j])
    y_set = frozenset(all_y)

    return dict(data=datas,x_set=x_set,y_set=y_set)
```

Get data from file

In [3]:

```
data_dict = get_data('EN/train')
```

(5 pts) - Write a function that estimates the emission parameters from the training set using MLE

(maximum likelihood estimation): e(x|y) = Count(y->x) / Count(y)

In [4]:

```
def get_emission_counts(data_dict):
    """
    returns (DataFrame,Series)
    an emission count (y->x) DataFrame and y count Series
    """
    data = data_dict['data']
    x_set = data_dict['x_set']
    y_set = data_dict['y_set']
    count_em_df = pd.DataFrame(np.zeros((len(x_set),len(y_set))),index=x_se
t,columns=y_set)
    count_y = pd.Series(np.zeros(len(y_set)),index=y_set)

    for instance in data:
        x_vector,y_vector = instance
        for i in range(len(x_vector)):
            x,y = x_vector[i],y_vector[i]
            count_em_df.loc[x,y]+=1
            count_y[y]+=1
    return count_em_df,count_y

def get_emission_params(data_dict):
    """
    returns DataFrame representing conditional probabilities P(y|x)
    """
    count_em_df,count_y = get_emission_counts(data_dict)
    return count_em_df/count_y
```

The following shows an example of the emission parameters that we get from the above functions:

In [5]:

```
em_df = get_emission_params(data_dict)
em_df.head()
```

Out[5]:

	I-negative	B-negative	O	B-positive	I-positive	I-neutral	B-neutral
1/26	0.0	0.0	0.000041	0.0	0.000000	0.0	0.0
ago	0.0	0.0	0.000083	0.0	0.000000	0.0	0.0
cod	0.0	0.0	0.000000	0.0	0.001647	0.0	0.0
Ruby	0.0	0.0	0.000041	0.0	0.000000	0.0	0.0
jobs	0.0	0.0	0.000041	0.0	0.000000	0.0	0.0

In [6]:

```
em_df.sum(axis=0)
```

Out[6]:

```
I-negative    1.0
B-negative    1.0
O             1.0
B-positive    1.0
I-positive    1.0
I-neutral     1.0
B-neutral     1.0
dtype: float64
```

(10 pts) One problem with estimating the emission parameters is that some words that appear in the test set do not appear in the training set. One simple idea to handle this issue is as follows. First, replace those words that appear less than k times in the training set with a special token #UNK# before training. This leads to a "modified training set". We then use such a modified training set to train our model.

During the testing phase, if the word does not appear in the "modified training set", we replace that word with #UNK# as well.

Set k to 3, implement this fix into your function for computing the emission parameters.

In [7]:

```
def get_modified_counts(data_dict,k):
    count_em_df,count_y = get_emission_counts(data_dict)

    counts_x = count_em_df.sum(axis=1)
    fail = counts_x[counts_x<k]

    unk = count_em_df.loc[fail.index].sum(axis=0)
```

```
    unk = count_em_df.loc[fail.index].sum(axis=0)
    unk.name = '#UNK#'

    modified_df = count_em_df.append(unk)
    modified_df = modified_df.drop(fail.index, axis=0)

    return modified_df,count_y


def get_modified_emission_params(data_dict,k=3):
    """
    returns DataFrame representing conditional probabilities P(y|x)
    """
    count_em_df,count_y = get_modified_counts(data_dict,k)
    return count_em_df/count_y
```

The following shows an example (specifically showing #UNK#) of the modified emission parameters that we get from the above functions:

In [8]:

```
modified_em_params = get_modified_emission_params(data_dict)
modified_em_params.tail()
```

Out[8]:

	I-negative	B-negative	O	B-positive	I-positive	I-neutral	B-neutral
blocks	0.000000	0.000000	0.000124	0.000000	0.000000	0.000000	0.000000
seems	0.000000	0.000000	0.000124	0.000000	0.000000	0.000000	0.000000
reservation	0.000000	0.000000	0.000289	0.000000	0.000000	0.000000	0.000000
neighborhood	0.000000	0.000000	0.000454	0.000828	0.000000	0.000000	0.000000
#UNK#	0.255639	0.183246	0.116492	0.242550	0.347611	0.217391	0.169231

In [9]:

```
modified_em_params.sum(axis=0)
```

Out[9]:

```
I-negative    1.0
B-negative    1.0
O             1.0
B-positive    1.0
I-positive    1.0
I-neutral     1.0
B-neutral     1.0
dtype: float64
```

In [10]:

```
em_df.loc[['four','NYC']]
```

Out[10]:

	I-negative	B-negative	O	B-positive	I-positive	I-neutral	B-neutral

four	0.0	0.0	0.000248	0.000828	0.0	0.000000	0.0
	I-negative	B-negative	O	B-positive	I-positive	I-neutral	B-neutral
NYC	0.0	0.0	0.000578	0.000000	0.0	0.043478	0.0

(10 pts) Implement a simple sentiment analysis system that produces the tag

```
y* = argmax_y e(x|y)
```

for each word x in the sequence.

In [20]:

```
def train(filename,k=3):
    data_dict = get_data(filename)
    return get_modified_emission_params(data_dict,k=k)

def argmax_y(emission_params,x):
    # check if x in trained x's
    if x not in emission_params.index:
        x = '#UNK#'
    p = emission_params.loc[x,:]

    max_p = None
    for col in p.index:
        if max_p is None:
            max_p = p.loc[col]
            y = col
        elif p.loc[col]>max_p:
            max_p = p.loc[col]
            y = col
    return y

def decode(filename,emission_params,outfile):
    f = open(filename,'r', encoding="utf8")
    lines = f.readlines()
    lines = [line.replace('\n','') for line in lines]
    #print lines

    for i in range(len(lines)):
        line = lines[i]
        if line != '':
            line = line +' '+argmax_y(emission_params,line)
        line += '\n'

        lines[i] = line

    fout = open(outfile,'w', encoding="utf8")
    for line in lines:
        fout.write(line)
    fout.close()
    print("decoding completed")
```

The following shows an example of a word and its corresponding tag produced:

In [12]:

```
word = 'NYC'

try:
    print(modified_em_params.loc[word])
except:
    word = '#UNK#'
    print(modified_em_params.loc[word])

argmax_y(modified_em_params,word)
```

```
I-negative    0.000000
B-negative    0.000000
O             0.000578
B-positive    0.000000
I-positive    0.000000
I-neutral     0.043478
B-neutral     0.000000
Name: NYC, dtype: float64
```

Out[12]:

```
'I-neutral'
```

Training and Decoding on EN data Results

In [22]:

```
emission_params = train('EN/train')
decode('EN/dev.in',emission_params,'EN/dev.p2.out')
```

```
decoding completed
```

```
>python3 evalResult.py EN/dev.out EN/dev.p2.out

#Entity in gold data: 226
#Entity in prediction: 1201

#Correct Entity : 165
Entity  precision: 0.1374
Entity  recall: 0.7301
Entity  F: 0.2313

#Correct Sentiment : 71
Sentiment  precision: 0.0591
Sentiment  recall: 0.3142
Sentiment  F: 0.0995
```

Training and Decoding on CN data Results

In [21]:

```
emission_params = train('CN/train')
decode('CN/dev.in',emission_params,'CN/dev.p2.out')
```

```
decoding completed
```

```
>python3 evalResult.py CN/dev.out CN/dev.p2.out

#Entity in gold data: 362
#Entity in prediction: 3318

#Correct Entity : 183
Entity  precision: 0.0552
Entity  recall: 0.5055
Entity  F: 0.0995

#Correct Sentiment : 57
Sentiment  precision: 0.0172
Sentiment  recall: 0.1575
Sentiment  F: 0.0310
```

Training and Decoding on FR data Results

In [23]:

```
emission_params = train('FR/train')
decode('FR/dev.in',emission_params,'FR/dev.p2.out')
```

```
decoding completed
```

```
>python3 evalResult.py FR/dev.out FR/dev.p2.out

#Entity in gold data: 223
#Entity in prediction: 1149

#Correct Entity : 182
Entity  precision: 0.1584
Entity  recall: 0.8161
Entity  F: 0.2653

#Correct Sentiment : 68
Sentiment  precision: 0.0592
Sentiment  recall: 0.3049
Sentiment  F: 0.0991
```

Training and Decoding on SG data Results

Note: Comment out the check formatting in get_data() to avoid assertion error.

```
Due to special case in SG data, where there are words with spaces (e
g; . . . . O)
```

In [37]:

```
emission_params = train('SG/train')
decode('SG/dev.in',emission_params,'SG/dev.p2.out')
```

```
decoding completed
```

```
>python3 evalResult.py SG/dev.out SG/dev.p2.out

#Entity in gold data: 1382
#Entity in prediction: 6542

#Correct Entity : 780
Entity  precision: 0.1192
Entity  recall: 0.5644
Entity  F: 0.1969

#Correct Sentiment : 311
Sentiment  precision: 0.0475
Sentiment  recall: 0.2250
Sentiment  F: 0.0785
```

Sentiment Analysis System using HMM (Continued)

In [1]:

```
import numpy as np
import pandas as pd
```

We now have a new function for processing unlabelled data (test data for our predictions).

In [2]:

```
def get_data(filename):
    f = open(filename, 'r', encoding="utf8")
    lines = f.readlines()
    datas = []

    start = 0
    for i in range(len(lines)):
        if lines[i] == '\n':
            datas.append(lines[start:i])
            start = i+1
        lines[i] = lines[i].replace('\n','')
        lines[i] = tuple(lines[i].split(' '))

    # check formatting
#     for i in range(len(datas)):
#         for j in range(len(datas[i])):
#             assert len(datas[i][j])==2


    for i in range(len(datas)):
        data = datas[i]
        x = [word[0] for word in data]
        y = [word[1] for word in data]
        datas[i] = [x,y]

    all_x = []
    for i in range(len(datas)):
        for j in range(len(datas[i][0])):
            all_x.append(datas[i][0][j])
    x_set = frozenset(all_x)

    all_y = []
    for i in range(len(datas)):
        for j in range(len(datas[i][0])):
            all_y.append(datas[i][1][j])
    y_set = frozenset(all_y)

    return dict(data=datas,x_set=x_set,y_set=y_set)

def get_unlabelled_data(filename):
    f = open(filename, 'r', encoding="utf8")
    lines = f.readlines()
```

```
    datas = []

    start = 0
    for i in range(len(lines)):
        if lines[i] == '\n':
            datas.append(lines[start:i])
            start = i+1
        lines[i] = lines[i].replace('\n','')

    return datas
```

(5 pts) Write a function that estimates the transition parameters from the training set using MLE

```
q(y_i|y_i-1) = Count(y_i-1, y_i) / Count(y_i-1)
```

The following special cases are also considered: q(STOP|y_n) and q(y_1|START).

In [3]:

```
data_dict = get_data('EN/train')
```

In [4]:

```
def get_transmission_params(data_dict):
    from_y = ['START'] + list(data_dict['y_set'])
    to_y = list(data_dict['y_set']) + ['STOP']
    l = len(from_y)
    transmission_count = pd.DataFrame(np.zeros((l,l)),index=from_y,columns=
to_y)

    datas = data_dict['data']
    for instance in datas:
        x_vector,y_vector = instance
        length = len(y_vector)
        for i in range(length+1):
            if i == 0 :
                transmission_count.loc['START',y_vector[0]] += 1

            elif i == length:
                transmission_count.loc[y_vector[i-1],'STOP'] +=1

            else:
                transmission_count.loc[y_vector[i-1],y_vector[i]] += 1

    y_count = transmission_count.sum(axis=1)
    transmission_params = transmission_count
    for i in range(len(transmission_count.index)):
        transmission_params.iloc[i,:] /=
transmission_params.iloc[i,:].sum()
    return transmission_params
```

The following shows the tranmission parameters that we get from using the above function:

In [5]:

```
trans_params = get_transmission_params(data_dict)
```

```
trans_params
```

Out[5]:

	I-positive	B-neutral	B-positive	I-negative	O	I-neutral	B-negative	STOP
START	0.000000	0.005339	0.043780	0.000000	0.940203	0.000000	0.010678	0.000000
I-positive	0.406919	0.000000	0.000000	0.000000	0.584843	0.000000	0.000000	0.008237
B-neutral	0.000000	0.000000	0.000000	0.000000	0.784615	0.200000	0.000000	0.015385
B-positive	0.298013	0.000000	0.000000	0.000000	0.688742	0.000000	0.000000	0.013245
I-negative	0.000000	0.000000	0.000000	0.398496	0.601504	0.000000	0.000000	0.000000
O	0.000000	0.002269	0.046448	0.000000	0.860119	0.000000	0.014933	0.076231
I-neutral	0.000000	0.000000	0.000000	0.000000	0.565217	0.434783	0.000000	0.000000
B-negative	0.000000	0.000000	0.000000	0.209424	0.782723	0.000000	0.000000	0.007853

In [6]:

```
trans_params.sum(axis=1)
```

Out[6]:

```
START          1.0
I-positive     1.0
B-neutral      1.0
B-positive     1.0
I-negative     1.0
O              1.0
I-neutral      1.0
B-negative     1.0
dtype: float64
```

(15 pts) Use the estimated transition and emission parameters, implement the Viterbi algorithm to compute the following (for a sentence with n words):

$$y^*_1,\ldots, y^*_n = \text{argmax}_{y1,\ldots,yn}(p)(x1,\ldots,xn,y1,\ldots,yn)$$

In [7]:

```
def get_emission_counts(data_dict):
    """
    returns (DataFrame,Series)
    an emission count (y->x) DataFrame and y count Series
    """
    data = data_dict['data']
    x_set = data_dict['x_set']
    y_set = data_dict['y_set']
    count_em_df = pd.DataFrame(np.zeros((len(x_set),len(y_set))),index=x_se
```

```
t,columns=y_set)
    count_y = pd.Series(np.zeros(len(y_set)),index=y_set)

    for instance in data:
        x_vector,y_vector = instance
        for i in range(len(x_vector)):
            x,y = x_vector[i],y_vector[i]
            count_em_df.loc[x,y]+=1
            count_y[y]+=1
    return count_em_df,count_y

def get_modified_counts(data_dict,k):
    count_em_df,count_y = get_emission_counts(data_dict)

    counts_x = count_em_df.sum(axis=1)
    fail = counts_x[counts_x<k]

    unk = count_em_df.loc[fail.index].sum(axis=0)
    unk.name = '#UNK#'

    modified_df = count_em_df.append(unk)
    modified_df = modified_df.drop(fail.index, axis=0)

    return modified_df,count_y


def get_modified_emission_params(data_dict,k=3):
    """
    returns DataFrame representing conditional probabilities P(y|x)
    """
    count_em_df,count_y = get_modified_counts(data_dict,k)
    return count_em_df/count_y
```

In [8]:

```
def vertibi(x_vector,trans_params,em_params):
    """
    x_vector: a list of string which represent the sequence of observations
    trans_params: a DataFrame with index from_states, columns to_states
                  containing the transmission probabilities
    em_params: a DataFrame with index observations, columns states which co
ntains
               the emission probabilities
    """

    states = trans_params.index.tolist()
    states.remove('START')
    states.remove('O')
    states = ['O']+states

    arr = np.zeros((len(states),len(x_vector))) *np.nan
    arr2 = np.zeros((len(states),len(x_vector))) *np.nan
    t1 = pd.DataFrame(arr,index=states,columns=x_vector)
    t2 = pd.DataFrame(arr2,index=states,columns=x_vector)

    for i in range(len(states)):
        t1.iloc[i,0] = trans_params.loc['START',t1.index[i]] * em_params.loc
[x_vector[0],t1.index[i]]
        t2.iloc[i,0] = 0
```

```
    for i in range(1,len(x_vector)):
        for j in range(len(states)):
            em_prob = em_params.loc[x_vector[i],states[j]] #prob of getting
x_i given state j
            maxx = None
            argmax = None
            for k in range(len(states)):
                prob = t1.iloc[k,i-1] * trans_params.loc[states[k],states[j]
] * em_prob
                if maxx is None:
                    argmax = k
                    maxx = prob

                elif prob>maxx:
                    argmax = k
                    maxx = prob

            t1.iloc[j,i] = maxx
            t2.iloc[j,i] = argmax


    for i in range(len(states)):
        prob = t1.iloc[i,len(x_vector)-1]
        t1.iloc[i,len(x_vector)-1] = t1.iloc[i,len(x_vector)-1] *
trans_params.loc[states[i],'STOP']

    prediction_indx = []
    maxx = None
    argmax = None
    for k in range(len(states)):
        prob = t1.iloc[k,len(x_vector)-1]
        if maxx is None:
            maxx = prob
            argmax = k
        elif prob > maxx:
            maxx = prob
            argmax = k

    prediction_indx.append(argmax)
    for i in range(len(x_vector)-1,0,-1):
        indx = t2.iloc[int(prediction_indx[0]),i]
        prediction_indx = [indx] + prediction_indx

    prediction = [states[int(i)] for i in prediction_indx]
    return prediction


def decode(fin,fout,trans_params,em_params):
    word_bag = em_params.index.tolist()
    unlabelled_datas = get_unlabelled_data(fin)

    results = []
    for obs_vector in unlabelled_datas:
        copy = []
        for i in range(len(obs_vector)):
            if obs_vector[i] in word_bag:
                copy.append(obs_vector[i])
            else:
                copy.append('#UNK#')
        result = vertibi(copy,trans_params,em_params)
```

```
        assert len(result) == len(obs_vector)
        results.append(result)

    fout = open(fout, 'w', encoding="utf8")
    for i in range(len(unlabelled_datas)):
        for j in range(len(unlabelled_datas[i])):
            x = unlabelled_datas[i][j]
            y = results[i][j]
            fout.write('{} {}\n'.format(x,y))
        fout.write('\n')
    fout.close
    print("vertibi decoding complete")
```

Training and Decoding on EN data Results

In [9]:

```
data_dict = get_data('EN/train')
a = get_transmission_params(data_dict)
b = get_modified_emission_params(data_dict,k=3)
decode('EN/dev.in','EN/dev.p3.out',a,b)
```

```
vertibi decoding complete
```

```
>python3 evalResult.py EN/dev.out EN/dev.p3.out

#Entity in gold data: 226
#Entity in prediction: 162

#Correct Entity : 104
Entity  precision: 0.6420
Entity  recall: 0.4602
Entity  F: 0.5361

#Correct Sentiment : 64
Sentiment  precision: 0.3951
Sentiment  recall: 0.2832
Sentiment  F: 0.3299
```

Training and Decoding on CN data Results

In [10]:

```
data_dict = get_data('CN/train')
a = get_transmission_params(data_dict)
b = get_modified_emission_params(data_dict,k=3)
decode('CN/dev.in','CN/dev.p3.out',a,b)
```

```
vertibi decoding complete
```

```
>python3 evalResult.py CN/dev.out CN/dev.p3.out
```

```
#Entity in gold data: 362
#Entity in prediction: 158

#Correct Entity : 64
Entity  precision: 0.4051
Entity  recall: 0.1768
Entity  F: 0.2462

#Correct Sentiment : 47
Sentiment  precision: 0.2975
Sentiment  recall: 0.1298
Sentiment  F: 0.1808
```

Training and Decoding on FR data Results

In [11]:

```
data_dict = get_data('FR/train')
a = get_transmission_params(data_dict)
b = get_modified_emission_params(data_dict,k=3)
decode('FR/dev.in','FR/dev.p3.out',a,b)
```

```
vertibi decoding complete
```

```
>python3 evalResult.py FR/dev.out FR/dev.p3.out

#Entity in gold data: 223
#Entity in prediction: 166

#Correct Entity : 112
Entity  precision: 0.6747
Entity  recall: 0.5022
Entity  F: 0.5758

#Correct Sentiment : 72
Sentiment  precision: 0.4337
Sentiment  recall: 0.3229
Sentiment  F: 0.3702
```

Training and Decoding on SG data Results

In [12]:

```
data_dict = get_data('SG/train')
a = get_transmission_params(data_dict)
b = get_modified_emission_params(data_dict,k=3)
decode('SG/dev.in','SG/dev.p3.out',a,b)
```

```
vertibi decoding complete
```

```
>python3 evalResult.py SG/dev.out SG/dev.p3.out

#Entity in gold data: 1382
#Entity in prediction: 723

#Correct Entity : 386
Entity  precision: 0.5339
Entity  recall: 0.2793
Entity  F: 0.3667

#Correct Sentiment : 244
Sentiment  precision: 0.3375
Sentiment  recall: 0.1766
Sentiment  F: 0.2318
```

Sentiment Analysis System (Continued)

(20 pts) Use the estimated transition and emission parameters, implement the alternative max-marginal decoding algorithm. Clearly describe the steps of your algorithm in your report.

Run the algorithm on the development sets EN/dev.in and FR/dev.in only. Write the outputs to EN/dev.p4.out and FR/dev.p4.out. Report the precision, recall and F scores for the outputs for both languages.

Hint: the max-marginal decoding involves the implementation of the forward-backward algorithm.

In [1]:

```
import numpy as np
import pandas as pd
```

Emission and Transition functions

In [2]:

```
def get_data(filename):
    f = open(filename,'r')
    lines = f.readlines()
    datas = []

    start = 0
    for i in range(len(lines)):
        if lines[i] == '\n':
            datas.append(lines[start:i])
            start = i+1
        lines[i] = lines[i].replace('\n','')
        lines[i] = tuple(lines[i].split(' '))

    # check formatting
    for i in range(len(datas)):
        for j in range(len(datas[i])):
            #print datas[i][j]
            assert len(datas[i][j])==2


    for i in range(len(datas)):
        data = datas[i]
        x = [word[0] for word in data]
        y = [word[1] for word in data]
        datas[i] = [x,y]

    all_x = []
    for i in range(len(datas)):
        for j in range(len(datas[i][0])):
            all_x.append(datas[i][0][j])
    x_set = frozenset(all_x)
```

```
    all_y = []
    for i in range(len(datas)):
        for j in range(len(datas[i][0])):
            all_y.append(datas[i][1][j])
    y_set = frozenset(all_y)

    return dict(data=datas,x_set=x_set,y_set=y_set)

def get_unlabelled_data(filename):
    f = open(filename,'r')
    lines = f.readlines()
    datas = []

    start = 0
    for i in range(len(lines)):
        if lines[i] == '\n':
            datas.append(lines[start:i])
            start = i+1
        lines[i] = lines[i].replace('\n','')

    return datas

def get_transmission_params(data_dict):
    from_y = ['START'] + list(data_dict['y_set'])
    to_y = list(data_dict['y_set']) + ['STOP']
    l = len(from_y)
    transmission_count = pd.DataFrame(np.zeros((l,l)),index=from_y,columns=
to_y)

    datas = data_dict['data']
    for instance in datas:
        x_vector,y_vector = instance
        length = len(y_vector)
        for i in range(length+1):
            if i == 0 :
                transmission_count.loc['START',y_vector[0]] += 1

            elif i == length:
                transmission_count.loc[y_vector[i-1],'STOP'] +=1

            else:
                transmission_count.loc[y_vector[i-1],y_vector[i]] += 1

    y_count = transmission_count.sum(axis=1)
    transmission_params = transmission_count
    for i in range(len(transmission_count.index)):
        transmission_params.iloc[i,:] /=
transmission_params.iloc[i,:].sum()
    return transmission_params

def get_emission_counts(data_dict):
    """
    returns (DataFrame,Series)
    an emission count (y->x) DataFrame and y count Series
    """
    data = data_dict['data']
    x_set = data_dict['x_set']
    y_set = data_dict['y_set']
    count_em_df = pd.DataFrame(np.zeros((len(x_set),len(y_set))),index=x_se
```

```
t,columns=y_set)
    count_y = pd.Series(np.zeros(len(y_set)),index=y_set)

    for instance in data:
        x_vector,y_vector = instance
        for i in range(len(x_vector)):
            x,y = x_vector[i],y_vector[i]
            count_em_df.loc[x,y]+=1
            count_y[y]+=1
    return count_em_df,count_y

def get_modified_counts(data_dict,k):
    count_em_df,count_y = get_emission_counts(data_dict)

    counts_x = count_em_df.sum(axis=1)
    fail = counts_x[counts_x<k]

    unk = count_em_df.loc[fail.index].sum(axis=0)
    unk.name = '#UNK#'

    modified_df = count_em_df.append(unk)
    modified_df = modified_df.drop(fail.index, axis=0)

    return modified_df,count_y


def get_modified_emission_params(data_dict,k=3):
    """
    returns DataFrame representing conditional probabilities P(y|x)
    """
    count_em_df,count_y = get_modified_counts(data_dict,k)
    return count_em_df/count_y
```

Max Marginal Functions

In [5]:

```
def get_forward_prob(x_vector,trans_params,em_params):

    states = trans_params.index.tolist()
    states.remove('START')
    states.remove('O')
    states = ['O']+states

    arr = np.zeros((len(states),len(x_vector))) *np.nan

    alpha = pd.DataFrame(arr,index=states,columns=x_vector)

    # base case
    for i in range(len(states)):
        alpha.iloc[i,0] = trans_params.loc['START',states[i]]

    # recursive case
    for i in range(1,len(x_vector)):
        for u in range(len(states)):
            summ = 0
            for v in range(len(states)):
                summ += alpha.iloc[v,i-1] * trans_params.loc[states[v],state
s[u]] * \
```

```
                        em_params.loc[x_vector[i],states[v]]
            alpha.loc[states[u],x_vector[i]] = summ

    return alpha

def get_backward_prob(x_vector,trans_params,em_params):

    states = trans_params.index.tolist()
    states.remove('START')
    states.remove('O')
    states = ['O']+states

    arr = np.zeros((len(states),len(x_vector))) *np.nan

    beta = pd.DataFrame(arr,index=states,columns=x_vector)

    # base case
    for i in range(len(states)):
        beta.iloc[i,len(x_vector)-1] = trans_params.loc[states[i],'STOP'] *
\
                                       em_params.loc[x_vector[-1],states[i]]

    # recursive case
    for i in range(len(x_vector)-2,-1,-1):
        for u in range(len(states)):
            summ = 0
            for v in range(len(states)):
                summ += beta.iloc[v,i+1] * trans_params.loc[states[u],states
[v]] * \
                        em_params.loc[x_vector[i],states[u]]
            beta.iloc[u,i] = summ

    return beta

def max_marginal_decode(x_vector,trans_params,em_params):

    alpha = get_forward_prob(x_vector,trans_params,em_params)
    beta = get_backward_prob(x_vector,trans_params,em_params)

    states = trans_params.index.tolist()
    states.remove('START')
    states.remove('O')
    states = ['O']+states

    prediction_indx = []

    for i in range(len(x_vector)):
        maxx = None
        argmax = None
        for u in range(len(states)):
            prob = alpha.iloc[u,i] * beta.iloc[u,i]
            if maxx is None:
                maxx = prob
                argmax = u
            elif prob > maxx:
                maxx = prob
                argmax = u
        prediction_indx.append(argmax)

    prediction = [states[indx] for indx in prediction_indx]
```

```
    return prediction

def decode_file(fin,fout,trans_params,em_params):
    word_bag = em_params.index.tolist()
    unlabelled_datas = get_unlabelled_data(fin)

    results = []
    for obs_vector in unlabelled_datas:
        copy = []
        for i in range(len(obs_vector)):
            if obs_vector[i] in word_bag:
                copy.append(obs_vector[i])
            else:
                copy.append('#UNK#')
        result = max_marginal_decode(copy,trans_params,em_params)
        assert len(result) == len(obs_vector)
        results.append(result)

    fout = open(fout,'w')
    for i in range(len(unlabelled_datas)):
        for j in range(len(unlabelled_datas[i])):
            x = unlabelled_datas[i][j]
            y = results[i][j]
            fout.write('{} {}\n'.format(x,y))
        fout.write('\n')
    fout.close
    print("max marginal decoding complete")
```

Training and Decoding on EN data Results

In [6]:

```
print('training ...')
data_dict = get_data('EN/train')
trans_params = get_transmission_params(data_dict)
em_params = get_modified_emission_params(data_dict,k=3)

print('decoding ...')
decode_file('EN/dev.in','EN/dev.p4.out',trans_params,em_params)
```

```
training ...
decoding ...
max marginal decoding complete
```

```
>python3 evalResult.py EN/dev.out EN/dev.p4.out

#Entity in gold data: 226
#Entity in prediction: 181

#Correct Entity : 94
Entity  precision: 0.5193
Entity  recall: 0.4159
Entity  F: 0.4619

#Correct Sentiment : 57
```

```
Sentiment  precision: 0.3149
Sentiment  recall: 0.2522
Sentiment  F: 0.2801
```

Training and Decoding on FR data Results

In [8]:

```
print('training ...')
data_dict = get_data('FR/train')
trans_params = get_transmission_params(data_dict)
em_params = get_modified_emission_params(data_dict,k=3)

print('decoding ...')
decode_file('FR/dev.in','FR/dev.p4.out',trans_params,em_params)
```

```
training ...
decoding ...
max marginal decoding complete
```

```
>python3 evalResult.py FR/dev.out FR/dev.p4.out

#Entity in gold data: 223
#Entity in prediction: 77

#Correct Entity : 26
Entity  precision: 0.3377
Entity  recall: 0.1166
Entity  F: 0.1733

#Correct Sentiment : 11
Sentiment  precision: 0.1429
Sentiment  recall: 0.0493
Sentiment  F: 0.0733
```

Challenge: An improved sentiment analysis system

(10 pts) Now, based on the training and development set, think of a better design for developing an improved sentiment analysis system for tweets using any model you like.

In the previous parts, we have adopted the use of the Hidden Markov Models (HMM) for the sentiment analysis system.

In this challenge, we have decided to use a different model for the system. The model we have decided to use is Conditional Random Fields (CRF).

A CRF can be considered as a generalization of HMM or we can say that a HMM is a particular case of CRF where constant probabilities are used to model state transitions.

In contrast to the generative model (HMM), CRF is a discriminative model and the primary advantage of CRFs over HMMs is their conditional nature, resulting in the relaxation of the independence assumptions required by HMMs

Import relevant libraries

NLTK is a leading platform for building Python programs to work with human language data and is great for natural language processing.

pycrfsuite is a python binding to CRFsuite - an implementation of Conditional Random Fields (CRFs) for labeling sequential data. https://github.com/scrapinghub/python-crfsuite

In [1]:

```
import nltk
import pycrfsuite
```

Prepare the Dataset for Training

We have two separate functions for obtaining data, one for the training data (labelled) and one for the test data (unlabelled).

In [2]:

```
def get_data(filename):
    f = open(filename,'r')
    lines = f.readlines()
    datas = []

    start = 0
    for i in range(len(lines)):
        if lines[i] == '\n':
```

```
            if lines[i] == '\n':
                datas.append(lines[start:i])
                start = i+1
            lines[i] = lines[i].replace('\n','')
            lines[i] = tuple(lines[i].split(' '))

    # check formatting
    for i in range(len(datas)):
        for j in range(len(datas[i])):
            assert len(datas[i][j])==2

    return datas

def get_unlabelled_data(filename):
    f = open(filename,'r')
    lines = f.readlines()
    datas = []

    start = 0
    for i in range(len(lines)):
        if lines[i] == '\n':
            datas.append(lines[start:i])
            start = i+1
        lines[i] = lines[i].replace('\n','')

    return datas
```

Generating Part-of-Speech Tags

First, we use a feature in NLP: Part-of-Speech (POS) tagging of the words. These tags indicate whether the word is a noun, verb or an adjective etc.

NLTK's POS tagger will be used to generate the POS tags for the data.

*Note that there is a slight difference in POS-tagging for labelled and unlabelled data.

In [3]:

```
def pos_tagging(datas):
    data = []

    for i, sentences in enumerate(datas):

        # Obtain the list of tokens in the data
        tokens = [t for t, label in sentences]

        # Perform POS tagging
        tagged = nltk.pos_tag(tokens)

        # Take the word, POS tag, and its label
        data.append([(w, POS, label) for (w, label), (word, POS) in zip(sent
ences, tagged)])

    return data

def unlabelled_pos_tagging(datas):
    data = []

    for i, sentences in enumerate(datas):
```

```
        # Obtain the list of tokens in the data
        tokens = [t for t in sentences]

        # Perform POS tagging
        tagged = nltk.pos_tag(tokens)

        # Take the word and its POS tag
        data.append(tagged)

    return data


"""
Example of data after POS-tagging
"""
datas = get_data('EN/train')
tagged_data = pos_tagging(datas)
print(tagged_data[0])

print("\n" + "BREAK" + "\n")

datas = get_unlabelled_data('EN/dev.in')
tagged_data = unlabelled_pos_tagging(datas)
print(tagged_data[0])
```

```
[('We', 'PRP', 'O'), ('were', 'VBD', 'O'), ('then', 'RB', 'O'), ('charged',
'VBN', 'O'), ('for', 'IN', 'O'), ('their', 'PRP$', 'O'), ('most', 'RBS', 'O
'), ('expensive', 'JJ', 'O'), ('sake', 'NN', 'O'), ('(', '(', 'O'), ('$', '
$', 'O'), ('20', 'CD', 'O'), ('+', 'NNP', 'O'), ('per', 'IN', 'O'),
('serving', 'VBG', 'O'), (')', ')', 'O'), ('when', 'WRB', 'O'), ('we', 'PRP
', 'O'), ('in', 'IN', 'O'), ('fact', 'NN', 'O'), ('drank', 'IN', 'O'), ('a'
, 'DT', 'O'), ('sake', 'NN', 'O'), ('of', 'IN', 'O'), ('less', 'JJR', 'O'),
('than', 'IN', 'O'), ('half', 'PDT', 'O'), ('that', 'DT', 'O'), ('price', '
NN', 'O'), ('.', '.', 'O')]

BREAK

[('When', 'WRB'), ('I', 'PRP'), ('called', 'VBD'), ('this', 'DT'), ('mornin
g', 'NN'), (',', ','), ('I', 'PRP'), ("didn't", 'VBP'), ('think', 'VB'), ('
I', 'PRP'), ('would', 'MD'), ('be', 'VB'), ('able', 'JJ'), ('to', 'TO'), ('
get', 'VB'), ('in', 'IN'), ('at', 'IN'), ('12', 'CD'), (',', ','), ('but',
'CC'), ('I', 'PRP'), ('was', 'VBD'), ('able', 'JJ'), ('to', 'TO'), ('get',
'VB'), ('in', 'IN'), (',', ','), ('along', 'IN'), ('with', 'IN'), ('four',
'CD'), ('other', 'JJ'), ('guests', 'NNS'), ('.', '.')]
```

Generating Features

POS tag is one of the features for each of the token. However, we require more features in the dataset for better accuracy.

We have adopted some of the more commonly used features for a word in named entity recognition:

```
The word (w) itself (converted to lowercase for normalisation)
The prefix/suffix of w (e.g. -ion)
The words surrounding w, such as the previous and the next word
Whether w is in uppercase or lowercase
Whether w is a number, or contains digits
```

```
    The POS tag of w, and those of the surrounding words
    Whether w is or contains a special character (e.g. hypen, dollar sig
    n)
```

In [4]:

```python
def word2features(sentence, i):
    word = sentence[i][0]
    POStag = sentence[i][1]

    # Common features for all words
    features = [
        'bias',
        'word.lower=' + word.lower(),
        'word[-3:]=' + word[-3:],
        'word[-2:]=' + word[-2:],
        'word.isupper=%s' % word.isupper(),
        'word.istitle=%s' % word.istitle(),
        'word.isdigit=%s' % word.isdigit(),
        'POStag=' + POStag
    ]

    # Features for words that are not
    # at the beginning of a sentence
    if i > 0:
        word1 = sentence[i-1][0]
        POStag1 = sentence[i-1][1]
        features.extend([
            '-1:word.lower=' + word1.lower(),
            '-1:word.istitle=%s' % word1.istitle(),
            '-1:word.isupper=%s' % word1.isupper(),
            '-1:word.isdigit=%s' % word1.isdigit(),
            '-1:POStag=' + POStag1
        ])
    else:
        # Indicate that it is the 'beginning of a sentence'
        features.append('BOS')

    # Features for words that are not
    # at the end of a sentence
    if i < len(sentence)-1:
        word1 = sentence[i+1][0]
        POStag1 = sentence[i+1][1]
        features.extend([
            '+1:word.lower=' + word1.lower(),
            '+1:word.istitle=%s' % word1.istitle(),
            '+1:word.isupper=%s' % word1.isupper(),
            '+1:word.isdigit=%s' % word1.isdigit(),
            '+1:POStag=' + POStag1
        ])
    else:
        # Indicate that it is the 'end of a document'
        features.append('EOS')

    return features
```

Train the model

Train the model

To train the model, we need to first prepare the training data and the corresponding labels.

We separate the data into two parts: features & labels - by extracting them from the data.

The process_data function serves as a form of convenience for getting our x_train and y_train.

In [5]:

```
# A function for extracting features in sentences
def extract_features(sentence):
    return [word2features(sentence, i) for i in range(len(sentence))]

# A function for generating the list of labels for each sentence
def get_labels(sentence):
    return [label for (token, POStag, label) in sentence]

# A function for getting x_train and y_train
def process_data(filename, labelled = True):
    if (labelled):
        datas = get_data(filename)
        tagged_data = pos_tagging(datas)
        x = [extract_features(sentence) for sentence in tagged_data]
        y = [get_labels(sentence) for sentence in tagged_data]
    else:
        datas = get_unlabelled_data(filename)
        tagged_data = unlabelled_pos_tagging(datas)
        x = [extract_features(sentence) for sentence in tagged_data]
        y = 0

    return x, y
```

Now, we train the model using pycrfsuite.Trainer

In [6]:

```
def train_model(filename, model_out):
    x_train, y_train = process_data(filename)

    # Set (verbose=True) to see the steps in training the model
    trainer = pycrfsuite.Trainer(verbose=False)

    # Submit training data to the trainer
    for xseq, yseq in zip(x_train, y_train):
        trainer.append(xseq, yseq)

    # Set the parameters of the model
    trainer.set_params({
        # coefficient for L1 penalty
        'c1': 0.1,

        # coefficient for L2 penalty
        'c2': 0.01,

        # maximum number of iterations
        'max_iterations': 200,

        # whether to include transitions that
        # are possible, but not observed
```

```
        'feature.possible_transitions': True
    })

    # model will be trained and output to the file as specified in the argu
ment
    trainer.train(model_out)
```

Apply CRF on Test Data

Using the trained model, we apply it to the test data to predict the labels for each word in the data.

In [7]:

```
def decode_file(fin, fout, model):
    x_test, y_test = process_data(fin, labelled=False)
    unlabelled_data = get_unlabelled_data(fin)

    tagger = pycrfsuite.Tagger()
    tagger.open(model)
    y_pred = [tagger.tag(xseq) for xseq in x_test]

    fout = open(fout,'w')
    for i in range(len(unlabelled_data)):
        for j in range(len(unlabelled_data[i])):
            x = unlabelled_data[i][j]
            y = y_pred[i][j]
            fout.write('{} {}\n'.format(x,y))
        fout.write('\n')
    fout.close
    print("Conditional Random Fields complete")
```

Training and Decoding on EN data results

In [8]:

```
print('training...')
trained_model = train_model('EN/train', 'crf_EN.model')

print('decoding ...')
decode_file('EN/dev.in','EN/dev.p5.out', 'crf_EN.model')
```

```
training...
decoding ...
Conditional Random Fields complete
```

```
>python3 evalResult.py EN/dev.out EN/dev.p5.out

#Entity in gold data: 226
#Entity in prediction: 179

#Correct Entity : 128
Entity  precision: 0.7151
Entity  recall: 0.5664
Entity  F: 0.6321
```

```
#Correct Sentiment : 87
Sentiment  precision: 0.4860
Sentiment  recall: 0.3850
Sentiment  F: 0.4296
```

Training and Decoding on FR data results

In [9]:

```
print('training...')
trained_model = train_model('FR/train', 'crf_FR.model')

print('decoding ...')
decode_file('FR/dev.in','FR/dev.p5.out', 'crf_FR.model')
```

```
training...
decoding ...
Conditional Random Fields complete
```

```
>python3 evalResult.py FR/dev.out FR/dev.p5.out

#Entity in gold data: 223
#Entity in prediction: 181

#Correct Entity : 144
Entity  precision: 0.7956
Entity  recall: 0.6457
Entity  F: 0.7129

#Correct Sentiment : 106
Sentiment  precision: 0.5856
Sentiment  recall: 0.4753
Sentiment  F: 0.5248
```

(10 pts) We will evaluate your system's performance on two held out test sets EN/test.in and FR/test.in. The test sets will only be released on 4 Dec 2017 at 5pm (48 hours before the deadline). Use your new system to generate the outputs. Write your outputs to EN/test.p5.out and FR/test.p5.out.

Training and Decoding on test(EN) data

In [10]:

```
print('training...')
trained_model = train_model('EN/train', 'crf_EN.model')
```

```
print('decoding ...')
decode_file('test/EN/test.in','test/EN/test.p5.out', 'crf_EN.model')
```

```
training...
decoding ...
Conditional Random Fields complete
```

Training and Decoding on test(FR) data

In [11]:

```
print('training...')
trained_model = train_model('FR/train', 'crf_FR.model')

print('decoding ...')
decode_file('test/FR/test.in','test/FR/test.p5.out', 'crf_FR.model')
```

```
training...
decoding ...
Conditional Random Fields complete
```