1. (a) $P(G=1 \mid B>0) = \frac{P(G=1, B>0)}{P(B>0)} = \frac{P(G=1, B=1)}{P(B>0)}$

$$= \frac{\frac{1}{2}}{\frac{1}{2}+\frac{1}{4}} = \frac{2}{3}$$

(b). $P(C_2 = G \mid C_1 = B) = \frac{P(C_2 = G, C_1 = B)}{P(C_1 = B)}$

$$= \frac{1/4}{1/2} = \frac{1}{2}$$

2. (a). The prosecutor is wrong, because he assume that

$P(\text{innocent} \mid \text{blood match}) = P(\text{blood match} \mid \text{innocent})$
$= 1\%$

so he claims $P(\text{guilty} \mid \text{blood match}) = 99\%$

But his assumption itself is wrong.

the equality exists iff $P(\text{innocent}) = \frac{1}{2}$

However, it's unknown, so his argument is wrong.

(b). The defender calculated $P(\text{guilty} \mid \text{blood match})$ regardless of $P(\text{guilty})$, so he is also wrong.

3. (a) $p(X_2 = H) = p(X_2 = H \mid X_1 = H)\, p(X_1 = H)$

$+ p(X_2 = H \mid X_1 = A)\, p(X_1 = A)$

$= 0.9 \times 1 + 0.1 \times 0 = 0.9$

(b). $p(Y_2 = \text{frown})$

$= P(Y_2 = \text{frown} \mid X_2 = H)\, P(X_2 = H)$

$+ P(Y_2 = \text{frown} \mid X_2 = A)\, P(X_2 = A)$

$= 0.1 \times 0.9 + 0.6 \times 0.1$

$= 0.15$

(c). $P(X_2 = H \mid Y_2 = \text{frown})$

$$= \frac{P(Y_2 = \text{frown} \mid X_2 = H)\, P(X_2 = H)}{\sum_{x_2} P(Y_2 = \text{frown} \mid X_2 = x_2)\, P(X_2 = x_2)}$$

$$= \frac{0.1 \times 0.9}{0.1 \times 0.9 + 0.6 \times 0.1} = \frac{9}{15} = 0.6$$

(d). $P(Y_{60} = \text{yell})$

$= P(Y_{60} = \text{yell} \mid X_{60} = A) \cdot P(X_{60} = A)$

$+ P(Y_{60} = \text{yell} \mid X_{60} = H)\, P(X_{60} = H)$

For $n \geq 2$,

$P(X_n = A) = P(X_n = A \mid X_{n-1} = A)\, P(X_{n-1} = A)$
$\quad + P(X_n = A \mid X_{n-1} = H)\, P(X_{n-1} = H)$

$P(X_n = H) = P(X_n = H \mid X_{n-1} = A)\, P(X_{n-1} = A)$
$\quad + P(X_n = H \mid X_{n-1} = H)\, P(X_{n-1} = H)$

Denote $\pi_n = \begin{pmatrix} P(X_n = A) \\ P(X_n = H) \end{pmatrix}$

$\pi_n = \varphi \cdot \pi_{n-1}$

$\pi_{60} = \varphi^{59} \pi_1 = \begin{pmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{pmatrix}^{59} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$

$= \begin{pmatrix} 0.50000096 \\ 0.49999904 \end{pmatrix}$

$P(Y_{60} = \text{yell}) \approx 0.1 \times 0.5 + 0.2 \times 0.5 = 0.15$

(b). In this problem, we don't assume $X_1 = H$

So we assume an initial distribution π_1,

$P(X_1 = x_1, \cdots X_5 = x_5 \mid Y_1 = \cdots = Y_5 = \text{frown})$

$$= \frac{P(Y_1 = \cdots = Y_5 = \text{frown} \mid X_1 = x_1 \cdots X_5 = x_5)\, P(X_1 = x_1 \cdots X_5 = x_5)}{P(Y_1 = \cdots = Y_5 = \text{frown})}$$

We are optimizing on $x_1 \cdots x_5$, so.

$P(Y_1 = \cdots = Y_5 = \text{frown})$ doesn't matter.

$$P(Y_1=\cdots=Y_5=\text{frown} \mid X_1=x_1,\cdots X_5=x_5)\,P(X_1=x_1\cdots X_5=x_5) = P(X_1=x_1)\prod_{t=2}^{5} P(X_t=x_t \mid X_{t-1}=x_{t-1})$$

$$\times \prod_{t=1}^{5} P(Y_t=\text{frown} \mid X_t=x_t)$$

$$= \pi_{1,x_1}\cdot\prod_{t=2}^{5}\alpha_{x_{t-1},x_t}\cdot\prod_{t=1}^{5}\psi_t(x_t)$$

Take log, we are maximizing:

$$\log\left(\psi_1(x_1)\cdot\pi_{1,x_1}\right) + \sum_{t=2}^{5}\log\left(\alpha_{x_{t-1},x_t}\cdot\psi_t(x_t)\right)$$

For $t \geq 2$.

$\alpha_{x_{t-1},x_t}\cdot\psi_t(x_t)$

- (H,H) $0.9 \times 0.1 = 0.09$
- (H,A) $0.1 \times 0.6 = 0.06$
- (A,H) $0.1 \times 0.1 = 0.01$
- (A,A) $0.9 \times 0.6 = 0.54$

so any states will goes to Angry for next step.

① If $x_1 = H$, the optimal result is:

$$\log(0.1\cdot\pi_{1H}) + \log(0.06 \times 0.54^3)$$

② if $x_1 = A$ the optimal result is.

$$\log(0.9\,\pi_{1A}) + \log(0.54^4)$$

$$\begin{cases} \pi_{1H} + \pi_{1A} = 1 \\ 0.01\ \pi_{1H} = 0.54\ \pi_{1A} \end{cases} \qquad \begin{cases} \pi_{1A} = \frac{1}{55} \\ \pi_{1H} = \frac{54}{55} \end{cases}$$

~~therefore~~ $\begin{cases} \text{if } P(X_1 = H) < \frac{54}{55} \end{cases}$

the best path will be (A, A, A, A, A)

$\begin{cases} \text{if } P(X_1 = H) > \frac{54}{55} \end{cases}$

the best path will be (H, A, A, A, A)

$\begin{cases} \text{if } P(X_1 = H) = \frac{54}{55}. \end{cases}$ both are equal.

1 Simulate 3.e with Viterbi Algorithm

In [1]:

```
states = ('Happy', 'Angry')
observations = ('smile', 'frown', 'laugh', 'yell')
transition_probability = {
   'Happy' : {'Happy': 0.9, 'Angry': 0.1},
   'Angry' :   {'Happy': 0.1, 'Angry': 0.9},
   }
emission_probability = {
   'Happy' : {'smile': 0.6, 'frown': 0.1, 'laugh': 0.2, 'yell':0.1},
   'Angry'   : {'smile': 0.1, 'frown': 0.6, 'laugh': 0.1,  'yell': 0.2},
   }

def Viterbit(obs, states, s_pro, t_pro, e_pro):
    path = { s:[] for s in states} # init path: path[s] represents the path ends with s
    curr_pro = {}
    for s in states:
        curr_pro[s] = s_pro[s]*e_pro[s][obs[0]]
    for i in range(1, len(obs)):
        last_pro = curr_pro
        curr_pro = {}
        for curr_state in states:
            max_pro, last_sta = max(((last_pro[last_state]*t_pro[last_state][curr_state]*e_pro[curr_
                                       for last_state in states))
            curr_pro[curr_state] = max_pro
            path[curr_state].append(last_sta)

    # find the final largest probability
    max_pro = -1
    max_path = None
    for s in states:
        path[s].append(s)
        if curr_pro[s] > max_pro:
            max_path = path[s]
            max_pro = curr_pro[s]
    return max_path


if __name__ == '__main__':
    obs = ['frown', 'frown', 'frown','frown','frown']
    # Test for the threshold
    p_list = [54/55 + 1e-5, 54/55 - 1e-5]
    for p in p_list:
        start_probability = {'Happy': p, 'Angry': 1-p}
        print(start_probability)
        print (Viterbit(obs, states, start_probability, transition_probability, emission_probability
```

executed in 20ms, finished 23:23:42 2019-09-15

```
{'Happy': 0.9818281818181818, 'Angry': 0.018171818181818233}
['Happy', 'Angry', 'Angry', 'Angry', 'Angry']
{'Happy': 0.9818081818181819, 'Angry': 0.018191818181818142}
['Angry', 'Angry', 'Angry', 'Angry', 'Angry']
```

In [1]:

```
import numpy as np
from collections import Counter

def data_loader():
    # load data
    with open('train','r') as file:
        train_data = file.read().split('\n')[:-1]
    with open('test','r') as file:
        test_data = file.read().split('\n')[:-1]
    return train_data, test_data

def parser(datum):
    # extract labels and words
    email_addr, label, words = datum.split(' ',2)
    words = words.split()
    # transform words into dictionary
    word_dict = dict(zip([words[i] for i in range(0, len(words), 2)], [int(words[i+1]) for i in ra
    # transform label into 0, 1
    if label == 'spam':
        label = 1
    else:
        label = 0
    return label, word_dict

def data_preprocessing(train_data, test_data):
    y_train = np.zeros(len(train_data))
    y_test = np.zeros(len(test_data))
    x_train = []
    x_test = []
    for i, datum in enumerate(train_data):
        label, word_dict = parser(datum)
        y_train[i] = label
        x_train.append(word_dict)
    for i, datum in enumerate(test_data):
        label, word_dict = parser(datum)
        y_test[i] = label
        x_test.append(word_dict)
    return x_train, y_train, x_test, y_test

def compute_prior(y_train):
    # compte prior distribution P(spam) and P(ham)
    ratio = Counter(y_train)
    return ratio[1]/len(y_train), ratio[0]/len(y_train)

def m_estimation_conditional_probability(x_train_frt, y_train, num_vocab, a):
    # compute P(w_j|spam) and P(w_j|ham)
    spam_idx = np.where(y_train == 1)[0]
    ham_idx = np.where(y_train == 0)[0]
    x_spam = x_train_frt[spam_idx, :]
    x_ham = x_train_frt[ham_idx, :]
    n_c = x_spam.sum(axis = 0)
    n = x_spam.sum()
    p = 1 / num_vocab
    m = num_vocab * a
    p_on_spam = (n_c + m*p) / (n+m)
    n_c = x_ham.sum(axis = 0)
    n = x_ham.sum()
    p_on_ham = (n_c + m*p) / (n+m)
    return p_on_spam, p_on_ham
```

```
def log_estimated_probability(p_spam, p_ham, p_on_spam_m, p_on_ham_m, x_frts):
    # compute log(P(spam, w_1, w_2,..., w_n)) and log(P(ham, w_1, w_2,..., w_n))
    p_spam_lookup = (x_frts > 0) * p_on_spam_m
    p_ham_lookup = (x_frts > 0) * p_on_ham_m
    p_spam_lookup[p_spam_lookup == 0] = 1
    p_ham_lookup[p_ham_lookup == 0] = 1
    log_p_spam = np.log(p_spam) + np.log(p_spam_lookup).sum(axis = 1)
    log_p_ham = np.log(p_ham) + np.log(p_ham_lookup).sum(axis = 1)
    return log_p_spam, log_p_ham

def accuarcy(y_true, y_pred):
    # calculate accuracy
    assert len(y_true) == len(y_pred)
    return (y_true==y_pred).sum()/len(y_true)
```

executed in 25ms, finished 23:23:32 2019-09-15

1 Load and preprocess data

In [2]:

```
# load data
train_data, test_data = data_loader()

# extract labels to 0,1 and features to dictinary
x_train, y_train, x_test, y_test = data_preprocessing(train_data, test_data)
```

executed in 978ms, finished 23:23:33 2019-09-15

2 Compute prior $P(\text{spam})$ and $P(\text{ham})$

In [3]:

```
# compute_prior
p_spam, p_ham = compute_prior(y_train)
print('Prior:')
print(p_spam, p_ham)
```

executed in 15ms, finished 23:23:33 2019-09-15

```
Prior:
0.5736666666666667 0.42633333333333334
```

Transform word dicts to feature vectors

In [4]:

```
from sklearn.feature_extraction import DictVectorizer
vectorizer = DictVectorizer(sparse=False)
x_train_frt = vectorizer.fit_transform(x_train)
x_test_frt = vectorizer.transform(x_test)
```

executed in 2.31s, finished 23:23:35 2019-09-15

3 Compute $P(w_j \mid spam)$ and $P(w_j \mid ham)$ by m-estimator

In [5]:

```
p_on_spam_m, p_on_ham_m = m_estimation_conditional_probability(x_train_frt, y_train, x_train_frt.sha
```

executed in 77ms, finished 23:23:35 2019-09-15

Top 5 spam word given spam

In [6]:

```
dict(zip(np.array(vectorizer.feature_names_)[np.array(np.argsort(p_on_spam_m)[::-1]).ravel()[:5]],
         p_on_spam_m[np.array(np.argsort(p_on_spam_m)[::-1]).ravel()[:5]]))
```

executed in 18ms, finished 23:23:35 2019-09-15

Out[6]:

```
{'enron': 0.03819438784473375,
 'a': 0.023618529446035274,
 'corp': 0.021737909849797956,
 'the': 0.02142517760233378,
 'to': 0.019687038335056983}
```

Top 5 spam word given ham

In [7]:

```
dict(zip(np.array(vectorizer.feature_names_)[np.array(np.argsort(p_on_ham_m)[::-1]).ravel()[:5]],
         p_on_ham_m[np.array(np.argsort(p_on_ham_m)[::-1]).ravel()[:5]]))
```

executed in 11ms, finished 23:23:35 2019-09-15

Out[7]:

```
{'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa': 0.0
44321708617828505,
 'enron': 0.04244372379371936,
 'the': 0.0331690756387682,
 'to': 0.025329676593911908,
 'a': 0.017736664090903333}
```

4 Predict and validation

Comparing $P(spam|w_1, w_2, \cdots, w_n)$ and $P(ham|w_1, w_2, \cdots, w_n)$ is equivalent to comparing $P(spam, w_1, w_2, \cdots, w_n)$ and $P(ham, w_1, w_2, \cdots, w_n)$. Therefore, $P(spam, w_1, w_2, \cdots, w_n)$ and $P(ham, w_1, w_2, \cdots, w_n)$ are compared to tell whether the email is spam or ham.

In [8]:

```
# compute log(P(spam, w_1, w_2,..., w_n)) and log(P(ham, w_1, w_2,..., w_n))
log_p_spam,log_p_ham = log_estimated_probability(p_spam, p_ham, p_on_spam_m, p_on_ham_m, x_test_frt)
test_pred = (log_p_spam > log_p_ham)
# compute accuracy
accuarcy(y_test, test_pred)
```

executed in 48ms, finished 23:23:35 2019-09-15

Out[8]:

0.908

5 Grid search for the best m

In [9]:

```
def pipeline(x_train_frt, y_train, x_test_frt, y_test, a):
    p_spam, p_ham = compute_prior(y_train)
    p_on_spam_m, p_on_ham_m = m_estimation_conditional_probability(x_train_frt, y_train, x_train_frt
    log_p_spam,log_p_ham = log_estimated_probability(p_spam, p_ham, p_on_spam_m, p_on_ham_m, x_test_
    test_pred = (log_p_spam > log_p_ham)
    print(str(a) + ":" + str(accuarcy(y_test, test_pred)))
```

executed in 10ms, finished 23:23:35 2019-09-15

In [10]:

```
a_grid = [1, 10,100,1000,10000]
for a in a_grid:
    pipeline(x_train_frt, y_train, x_test_frt, y_test, a)
```

executed in 685ms, finished 23:23:36 2019-09-15

```
1:0.908
10:0.911
100:0.916
1000:0.863
10000:0.778
```

We have the highest accuarcy at $m = 100$. For m small, the impact of prior is weak, $P(w_j \mid spam)$ are dominated by n_c/n. This might leads to easy overfit. For m large, the impact of prior is strong, $P(w_j \mid spam)$ dominated by p. In this case, different word won't have different impact on the final decision, which may leads to underfit. Therefore, m can be neither too larger nor to small, and our experiment also indicate that $m = 100$ is a good hyperparameter.

6 How to beat the classifier?

I will try to paraphrase words with high $P(w_j \mid spam)$ and low $P(w_j \mid ham)$ in the email with some other words with low $P(w_j \mid spam)$ or high $P(w_j \mid ham)$. If the core idea of the email made some words with high $P(w_j \mid spam)$ or low $P(w_j \mid ham)$ inevtiable. I would add redundant sentences with words have low $P(w_j \mid spam)$ or high $P(w_j \mid ham)$ to weaken the effect of bad words.