In [9]:

```
import os
import numpy as np
import random
import math
from collections import Counter
```

In [10]:

```
def make_Dictionary(train_dir):
    emails = [os.path.join(train_dir,f) for f in os.listdir(train_dir)]
    all_words = []
    for mail in emails:
        with open(mail) as m:
            for i,line in enumerate(m):
                if i == 2:  #Body of email is only 3rd line of text file
                    words = line.split()
                    all_words += words
    dictionary = Counter(all_words)
    dictionary_temp = Counter(all_words)

    # Paste code for non-word removal here(code snippet is given below)
    list_to_remove = dictionary_temp.keys()
    for item in list_to_remove:
        if item.isalpha() == False:    #Determine whether it is punctuation
            del dictionary[item]
        elif len(item) == 1:           #
            del dictionary[item]
    dictionary = dictionary.most_common(3000)
    return dictionary
```

In [11]:

```
def extract_features(root_dir,dictionary):
    emails = [os.path.join(root_dir,f) for f in os.listdir(root_dir)]
    all_words = []
    features_matrix = np.zeros((len(emails),3000))
    docID = 0
    for mail in emails:
        with open(mail) as m:
            for i,line in enumerate(m):
                if i == 2:
                    words = line.split()
                    for word in words:
                        wordID = 0
                        for i,d in enumerate(dictionary):
                            if d[0] == word:
                                wordID = i
                                features_matrix[docID,wordID] = words.count(word)
        docID = docID + 1
    return features_matrix
```

In [12]:

```
Path = "train-mails"
dir = make_Dictionary(Path)
# print('finish dir:\n', dir)
train_matrix = extract_features(Path,dir)
# print(train_matrix)

Path_test = "test-mails"
test_matrix = extract_features(Path_test,dir)
# print(test_matrix)
```

In [46]:

```
class NaiveBayes:
    train_data = []
    Mean = {}
    Std = {}
    Prob = {}
    def fit(self, x, y):
        #divide x by x
        DivideX = {}
        for i in range(len(y)):
            if (y[i] not in DivideX):
                DivideX[y[i]]=[]
                DivideX[y[i]].append(x[i])
            else:
                DivideX[y[i]] = np.vstack((DivideX[y[i]], x[i]))
        self.train_data = DivideX
        for key in DivideX:
            self.Mean[key] = []
            self.Std[key] = []
            self.Prob[key] = []
            for i in range(DivideX[key].shape[1]):
                self.Mean[key].append(np.mean(DivideX[key][:, i]))
                #Divide
                c = 0
                for j in DivideX[key][:, i]:
                    if j >= self.Mean[key][i]:
                        c += 1
                num = len(DivideX[key][:, i])
                self.Prob[key].append(c + 1/num + 1)
                #
                if np.std(DivideX[key][:, i]) == 0:
                    self.Std[key].append(0.00001)
                else:
                    self.Std[key].append(np.std(DivideX[key][:, i]))
        # print(self.Mean)
    def probabilty(self, data, key):
        prob = 1
        # print(data)
        for i in range(len(data)):
            if self.Std[key][i] == 0:
                if data[i] >= self.Mean[key][i]:
                    prob *= self.Prob[key][i]
                else:
                    prob *= (1 - self.Prob[key][i])
            else:
                exp = math.exp(-(math.pow(data[i] - self.Mean[key][i], 2)/(2*math.pow(self.Std[ke
y][i], 2))))
                prob *= (1/(math.sqrt(2*math.pi)*self.Std[key][i])) * exp
        return prob
    def predict(self, data):
        result = []
        for i in range(data.shape[0]):
            MaxProb = 0
            MaxProb_key = 0
            for key in self.Mean:
                Prob_temp = self.probabilty(data[i], key)
                if(Prob_temp > MaxProb):
                    MaxProb = Prob_temp
                    MaxProb_key = key
            result.append(MaxProb_key)
```

```
        return result
    def accuracy(self, data, prediction):
        correct = 0
        for x in range(len(data)):
            if data[x] == prediction[x]:
                correct += 1
        return (correct/float(len(data)))
    def ConfusionMatrix(self, data, prediction):
        TP = 0
        TN = 0
        FP = 0
        FN = 0
        for i in range(len(data)):
            if data[i] == 1 and prediction[i] == 1:
                TP += 1
            if data[i] == 1 and prediction[i] == 0:
                FP += 1
            if data[i] == 0 and prediction[i] == 1:
                FN += 1
            if data[i] == 0 and prediction[i] == 0:
                TN += 1
        return TP, TN, FP, FN
    def Precision(self, data, prediction):
        TP, TN, FP, FN = self.ConfusionMatrix(data, prediction)
        return TP/(TP + FP)
    def Recall(self, data, prediction):
        TP, TN, FP, FN = self.ConfusionMatrix(data, prediction)
        return TP/(TP + FN)
    def F1_score(self, data, prediction, Beta = 1):
        TP, TN, FP, FN = self.ConfusionMatrix(data, prediction)
        return (1 + Beta ** 2)*self.Precision(data, prediction) * self.Recall(data, prediction) /
(Beta ** 2 * self.Precision(data, prediction) + self.Recall(data, prediction))
```

In [47]:

```
NB = NaiveBayes()
train_labels = np.zeros(train_matrix.shape[0])
train_labels[351:701] = 1
NB.fit(train_matrix, train_labels)
pred = NB.predict(train_matrix)
# print(NB.accuracy(train_labels, pred))
print('train data:')
train_accuracy = NB.accuracy(train_labels, pred)
train_TP, train_TN, train_FP, train_FN = NB.ConfusionMatrix(train_labels, pred)
train_Precision = NB.Precision(train_labels, pred)
train_Recall = NB.Recall(train_labels, pred)
train_F1score = NB.F1_score(train_labels, pred)
print('train_accuracy:{}'.format(train_accuracy))
print('train_Precision:{}'.format(train_Precision))
print('train_Recall:{}'.format(train_Recall))
print('train_F1_score:{}'.format(train_F1score))
```

```
train data:
train_accuracy:0.8988603988603988
train_Precision:0.7971428571428572
train_Recall:1.0
train_F1_score:0.8871224165341812
```

In [48]:

```
pred_test = NB.predict(test_matrix)
test_labels = np.zeros(test_matrix.shape[0])
test_labels[130:] = 1
# print(pred_test)
# print(NB.accuracy(test_label,pred_test))

Test_TP,Test_TN,Test_FP,Test_FN = NB.ConfusionMatrix(test_labels,pred_test)
print('test data:')
test_accuracy = NB.accuracy(test_labels,pred_test)
test_TP,test_TN,test_FP,test_FN = NB.ConfusionMatrix(test_labels,pred)
test_Precision = NB.Precision(test_labels,pred_test)
test_Recall = NB.Recall(test_labels,pred_test)
test_F1score = NB.F1_score(test_labels,pred_test)
print('test_accuracy:{}'.format(test_accuracy))
print('test_Precision:{}'.format(test_Precision))
print('test_Recall:{}'.format(test_Recall))
print('test_F1_score:{}'.format(test_F1score))
```

```
test data:
test_accuracy:0.8038461538461539
test_Precision:0.6076923076923076
test_Recall:1.0
test_F1_score:0.7559808612440191
```

This assignment using Gaussian Naïve Bayes classifier to finish.In this task ,the biggest problem encountered is that the variance of some variables is 0. In the final result, we use 0,00001 as the vcariance of a variable which variance is 0. We also tried to use discreate prior probabilities to treat variables with a variance of 0 as Bernoulli distrubutions. Treat variable balues greater than the mean as 1 and less than the mean as 0. And make the probability non-zero through Laplace transform, though we are not reach the better result.

In the final result, our naïve bayes algorithm achieves 80% accuracy, and the Precision is 60.7%, Recall is 1, F1-score is 76%. Compare Naïve Bayes of sklearn, my algorithm is much worse. I think the flaw is mainly in the way of dealing with attribute which variance is zero.