Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel$\rightarrow$Restart) and then **run all cells** (in the menubar, select Cell$\rightarrow$Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:

```
NAME = "Ayush Koirala"
ID = "st122802"
```

Lab 06: Generative classifiers: Naive Bayes

As discussed in class, a Naive Bayes classifier works as follows: $$\begin{eqnarray} p(y \mid \mathbf{x} ; \theta) & = & \frac{p(\mathbf{x} \mid y ; \theta) p(y ; \theta)} {p(\mathbf{x} ; \theta)} \\ & \propto & p(\mathbf{x} \mid y ; \theta) p(y ; \theta) \\ & \approx & p(y ; \theta) \prod_j p(x_j \mid y ; \theta) \end{eqnarray}$$ We will use Naive Bayes to perform diabetes diagnosis and text classification.

Example 1: Diabetes classification

In this example we predict wheter a patient with specific diagnostic measurements has diabetes or not. As the features are continuous, we will model the conditional probabilities $p(x_j \mid y ; \theta)$ as univariate Gaussians with mean $\mu_{j,y}$ and standard deviation $\sigma_{j,y}$.

The data are originally from the U.S. National Institute of Diabetes and Digestive and Kidney Diseases (NIDDK) and are available from Kaggle (https://www.kaggle.com/uciml/pima-indians-diabetes-database)

In [1]:

```
import csv
import math
import random
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
```

Data manipulation

First we have some functions to read the dataset, split it into train and test, and partition it according to target class (y).

In [2]:

```
# Load data from CSV file
def loadCsv(filename):
    data_raw = pd.read_csv(filename)
    headers = data_raw.columns
    dataset = data_raw.values
    return dataset, headers

# Split dataset into test and train with given ratio
def splitDataset(test_size,*arrays,**kwargs):
    return train_test_split(*arrays,test_size=test_size,**kwargs)

# Separate training data according to target class
# Return key value pairs array in which keys are possible target variable values
# and values are the data records.

def data_split_byClass(dataset):
    Xy = {}
    for i in range(len(dataset)):
        datapair = dataset[i]
        # datapair[-1] (the last column) is the target class for this record.
        # Check if we already have this value as a key in the return array
        if (datapair[-1] not in Xy):
            # Add class as key
            Xy[datapair[-1]] = []
        # Append this record to array of records for this class key
        Xy[datapair[-1]].append(datapair)
    return Xy
```

Model training

Next we have some functions used for training the model. Parameters include mean and standard deviation, used to partition numerical variables into categorical variables, as well as

In [3]:

```
# Parameters of a Gaussian are its mean and standard deviation

def mean(numbers):
    return sum(numbers)/float(len(numbers))

def stdev(numbers):
    avg = mean(numbers)
    variance = sum([pow(x-avg,2) for x in numbers])/float(len(numbers)-1)
    return math.sqrt(variance)

# Calculate Gaussian parameters mu and sigma for each attribute over a dataset

def get_gaussian_parameters(X,y):
    parameters = {}
    unique_y = np.unique(y)
    for uy in unique_y:
        mean = np.mean(X[y==uy],axis=0)
        std = np.std(X[y==uy],axis=0)
        py = y[y==uy].size/y.size
        parameters[uy] = {'prior':py,'mean':mean,'std':std}
    return parameters, unique_y

def calculateProbability(x, mu, sigma):
    sigma = np.diag(sigma**2)
    x = x.reshape(-1,1)
    mu = mu.reshape(-1,1)
    exponent = np.exp(-1/2*(x-mu).T@np.linalg.inv(sigma)@(x-mu))
    return ((1/(np.sqrt(((2*np.pi)**x.size)*np.linalg.det(sigma))))*exponent)[0,0]
```

Model testing

Next some functions for testing the model on a test set and computing its accuracy. Note that we assume $$ p(y \mid \mathbf{x} ; \theta) \propto p(\mathbf{x} \mid y ; \theta), $$ which means we assume that the priors $p(y)$ are equal for each possible value of y.

In [4]:

```
# Calculate class conditional probabilities for given input data vector

def predict_one(x,parameters,unique_y,prior = True):
    probabilities = []
    for key in parameters.keys():
        probabilities.append(calculateProbability(x,parameters[key]['mean'],parameters[key]['std'])*(parameters[key]['prior']**(flo
at(prior))))
    probabilities = np.array(probabilities)
    return unique_y[np.argmax(probabilities)]

def getPredictions(X, parameters, unique_y,prior=True):
    predictions = []
    for i in range(X.shape[0]):
        predictions.append(predict_one(X[i],parameters,unique_y,prior))
    return np.array(predictions)

# Get accuracy for test set

def getAccuracy(y, y_pred):
    correct = len(y[y==y_pred])
    return correct/y.size
```

Experiment

Here we load the diabetes dataset, split it into training and test data, train a Gaussian NB model, and test the model on the test set.

In [5]:

```
# Load dataset

filename = 'diabetes.csv'
dataset, headers = loadCsv(filename)
#print(headers)
#print(np.array(dataset)[0:5,:])

# Split into training and test

X_train,X_test,y_train,y_test = splitDataset(0.4,dataset[:,:-1],dataset[:,-1])
print("Total =",len(dataset),"Train =", len(X_train),"Test =",len(X_test))

# Train model

parameters, unique_y = get_gaussian_parameters(X_train,y_train)
prediction = getPredictions(X_test,parameters,unique_y)
print("Accuracy with Prior =",getAccuracy(y_test,prediction))

# Test model

prediction = getPredictions(X_test,parameters,unique_y,prior = False)
print("Accuracy without Prior =",getAccuracy(y_test,prediction))
```

```
Total = 768 Train = 460 Test = 308
Accuracy with Prior = 0.7564935064935064
Accuracy without Prior = 0.737012987012987
```

Exercise In lab / take home work (20 points)

Find out the proportion of the records in your dataset are positive vs. negative. Can we conclude that $p(y=1) = p(y=0)$? If not, add the priors $p(y=1)$ and $p(y=0)$ to your NB model. Does it improve the result?

In [6]:

```
# YOUR CODE HERE
#raise NotImplementedError()
for key,value in parameters.items():
    print(f"The prior, p(y={key}): {parameters[key]['prior']}")
```

```
The prior, p(y=0.0): 0.6695652173913044
The prior, p(y=1.0): 0.33043478260869563
```

Explain that you can conclude that $p(y=1) = p(y=0)$? If not, add the priors $p(y=1)$ and $p(y=0)$ to your NB model. Does it improve the result? (double click to explain)

The output of the above code shows that p(y=0) = 0.669 and p(y=1) = 0.33304 are not equal. When we compare the accuracy of our model with and without prior consideration in calculating probability, we can see that the accuracy with prior is greater than the accuracy without prior. This means that taking the prior into account improves the outcome.

If p(y=0) and p(y=1) are equal, including priors in the prediction has no effect on accuracy because the probability p(y|x;theta) for both classes was multiplied by the same number. It's the same as scaling both classes' probality by the same constants. However, if the priors are unequal, or if the training dataset is not balanced (equal number of training samples for each class), the priors become biased more towards the class that occurs most frequently in the training samples. So, if we do not consider such priors to predict, it is equivalent to ignoring the probability of the test sample of such class occurring in the test dataset, which we assume to be the case.

But if the probalility of occurence of the test sample of a class in the test dataset completely differed from that of training datasets then using the prior may result in poor performance. In such case you may not want to remove such prior from the model, meaning not using the prior to classify the test sets.

Example 2: Text classification

This example has been adapted from a post by Jaya Aiyappan, available at Analytics Vidhya (https://medium.com/analytics-vidhya/naive-bayes-classifier-for-text-classification-556fabaf252b#:~:text=The%20Naive%20Bayes%20classifier%20is,time%20and%20less%20training%20data).

We will generate a small dataset of sentences that are classified as either "statements" or "questions."

We will assume that occurance and placement of words within a sentence is independent of each other (i.e., the features are conditionally independent given y). So the sentence "this is my book" is the same as "is this my book." We will treat words as case insensitive.

In [ ]:

In [7]:

```
# Generate text data for two classes, "statement" and "question"

text_train = [['This is my novel book', 'statement'],
              ['this book has more than one author', 'statement'],
              ['is this my book', 'question'],
              ['They are novels', 'statement'],
              ['have you read this book', 'question'],
              ['who is the novels author', 'question'],
              ['what are the characters', 'question'],
              ['This is how I bought the book', 'statement'],
              ['I like fictional characters', 'statement'],
              ['what is your favorite book', 'question']]

text_test = [['this is the book', 'statement'],
             ['who are the novels characters', 'question'],
             ['is this the author', 'question'],
            ['I like apples']]

# Load training and test data into pandas data frames

training_data = pd.DataFrame(text_train, columns= ['sentence', 'class'])
print(training_data)
print('\n-----------------------------------------\n')
testing_data = pd.DataFrame(text_test, columns= ['sentence', 'class'])
print(testing_data)
```

```
                           sentence      class
0              This is my novel book  statement
1  this book has more than one author  statement
2                     is this my book   question
3                     They are novels  statement
4            have you read this book   question
5           who is the novels author   question
6            what are the characters   question
7      This is how I bought the book  statement
8        I like fictional characters  statement
9         what is your favorite book   question

-----------------------------------------

                       sentence      class
0              this is the book  statement
1  who are the novels characters   question
2            is this the author   question
3                 I like apples       None
```

In [8]:

```
# Partition training data by class

stmt_docs = [train['sentence'] for index,train in training_data.iterrows() if train['class'] == 'statement']
question_docs = [train['sentence'] for index,train in training_data.iterrows() if train['class'] == 'question']
all_docs = [train['sentence'] for index,train in training_data.iterrows()]

# Get word frequencies for each sentence and class

def get_words(text):
    # Initialize word list
    words = [];
    # Loop through each sentence in input array
    for text_row in text:
        # Check the number of words. Assume each word is separated by a blank space
        # so that the number of words is the number of blank spaces + 1
        number_of_spaces = text_row.count(' ')
        # loop through the sentence and get words between blank spaces.
        for i in range(number_of_spaces):
            # Check for for last word
            words.append([text_row[:text_row.index(' ')].lower()])
            text_row = text_row[text_row.index(' ')+1:]
            i = i + 1
        words.append([text_row])
    return np.unique(words)

# Get frequency of each word in each document

def get_doc_word_frequency(words, text):
    word_freq_table = np.zeros((len(text),len(words)), dtype=int)
    i = 0
    for text_row in text:
        # Insert extra space between each pair of words to prevent
        # partial match of words
        text_row_temp = ''
        for idx, val in enumerate(text_row):
            if val == ' ':
                 text_row_temp = text_row_temp + '  '
            else:
                  text_row_temp = text_row_temp + val.lower()
        text_row = ' ' + text_row_temp + ' '
        j = 0
        for word in words:
            word = ' ' + word + ' '
            freq = text_row.count(word)
            word_freq_table[i,j] = freq
```

```
            j = j + 1
        i = i + 1

    return word_freq_table
```

In [9]:

```
# Get word frequencies for statement documents

word_list_s = get_words(stmt_docs)
word_freq_table_s = get_doc_word_frequency(word_list_s, stmt_docs)
tdm_s = pd.DataFrame(word_freq_table_s, columns=word_list_s)
print(tdm_s)
```

```
   are  author  book  bought  characters  fictional  has  how  i  is  like  \
0    0       0     1       0           0          0    0    0  0   1     0
1    0       1     1       0           0          0    1    0  0   0     0
2    1       0     0       0           0          0    0    0  0   0     0
3    0       0     1       1           0          0    0    1  1   1     0
4    0       0     0       0           1          1    0    0  1   0     1

   more  my  novel  novels  one  than  the  they  this
0     0   1      1       0    0     0    0     0     1
1     1   0      0       0    1     1    0     0     1
2     0   0      0       1    0     0    0     1     0
3     0   0      0       0    0     0    1     0     1
4     0   0      0       0    0     0    0     0     0
```

In [10]:

```
# Get word frequencies over all statement documents

freq_list_s = word_freq_table_s.sum(axis=0)
freq_s = dict(zip(word_list_s,freq_list_s))
print(freq_s)
```

```
{'are': 1, 'author': 1, 'book': 3, 'bought': 1, 'characters': 1, 'fictional': 1, 'has': 1, 'how': 1, 'i': 2, 'is': 2,
'like': 1, 'more': 1, 'my': 1, 'novel': 1, 'novels': 1, 'one': 1, 'than': 1, 'the': 1, 'they': 1, 'this': 3}
```

In [11]:

```
# Get word frequencies for question documents

word_list_q = get_words(question_docs)
word_freq_table_q = get_doc_word_frequency(word_list_q, question_docs)
tdm_q = pd.DataFrame(word_freq_table_q, columns=word_list_q)
print(tdm_q)
```

```
   are  author  book  characters  favorite  have  is  my  novels  read  the  \
0    0       0     1           0         0     0   1   1       0     0    0
1    0       0     1           0         0     1   0   0       0     1    0
2    0       1     0           0         0     0   1   0       1     0    1
3    1       0     0           1         0     0   0   0       0     0    1
4    0       0     1           0         1     0   1   0       0     0    0

   this  what  who  you  your
0     1     0    0    0     0
1     1     0    0    1     0
2     0     0    1    0     0
3     0     1    0    0     0
4     0     1    0    0     1
```

In [12]:

```
# Get word frequencies over all question documents

freq_list_q = word_freq_table_q.sum(axis=0)
freq_q = dict(zip(word_list_q,freq_list_q))
print(freq_q)
print(freq_list_s)
print(freq_list_q)
```

```
{'are': 1, 'author': 1, 'book': 3, 'characters': 1, 'favorite': 1, 'have': 1, 'is': 3, 'my': 1, 'novels': 1, 'read':
1, 'the': 2, 'this': 2, 'what': 2, 'who': 1, 'you': 1, 'your': 1}
[1 1 3 1 1 1 1 1 2 2 1 1 1 1 1 1 1 1 3]
[1 1 3 1 1 1 3 1 1 1 2 2 2 1 1 1]
```

In [13]:

```
# Get word probabilities for statement class
a = 1
prob_s = []
for count in freq_list_s:
    #print(word, count)
    prob_s.append((count+a)/(sum(freq_list_s)+len(freq_list_s)*a))
prob_s.append(a/(sum(freq_list_s)+len(freq_list_s)*a))

# Get word probabilities for question class

prob_q = []
for count in freq_list_q:
    prob_q.append((count+a)/(sum(freq_list_q)+len(freq_list_q)*a))
prob_q.append(a/(sum(freq_list_q)+len(freq_list_q)*a))


print('Probability of words for "statement" class \n')
print(dict(zip(word_list_s, prob_s)))
print('------------------------------------------ \n')
print('Probability of words for "question" class \n')
print(dict(zip(word_list_q, prob_q)))
```

```
Probability of words for "statement" class

{'are': 0.043478260869565216, 'author': 0.043478260869565216, 'book': 0.08695652173913043, 'bought': 0.043478260869565
216, 'characters': 0.043478260869565216, 'fictional': 0.043478260869565216, 'has': 0.043478260869565216, 'how': 0.0434
78260869565216, 'i': 0.06521739130434782, 'is': 0.06521739130434782, 'like': 0.043478260869565216, 'more': 0.043478260
869565216, 'my': 0.043478260869565216, 'novel': 0.043478260869565216, 'novels': 0.043478260869565216, 'one': 0.0434782
60869565216, 'than': 0.043478260869565216, 'the': 0.043478260869565216, 'they': 0.043478260869565216, 'this': 0.086956
52173913043}
------------------------------------------

Probability of words for "question" class

{'are': 0.05128205128205128, 'author': 0.05128205128205128, 'book': 0.10256410256410256, 'characters': 0.0512820512820
5128, 'favorite': 0.05128205128205128, 'have': 0.05128205128205128, 'is': 0.10256410256410256, 'my': 0.051282051282051
28, 'novels': 0.05128205128205128, 'read': 0.05128205128205128, 'the': 0.07692307692307693, 'this': 0.0769230769230769
3, 'what': 0.07692307692307693, 'who': 0.05128205128205128, 'you': 0.05128205128205128, 'your': 0.05128205128205128}
```

In [14]:

```
# Calculate prior for one class

def prior(className):
    denominator = len(stmt_docs) + len(question_docs)

    if className == 'statement':
        numerator =  len(stmt_docs)
    else:
        numerator =  len(question_docs)

    return np.divide(numerator,denominator)

# Calculate class conditional probability for a sentence

def classCondProb(sentence, className):
    words = get_words(sentence)
    prob = 1
    for word in words:
        if className == 'statement':
            idx = np.where(word_list_s == word)
            prob = prob * prob_s[np.array(idx)[0,0]]
        else:
            idx = np.where(word_list_q == word)
            prob = prob * prob_q[np.array(idx)[0,0]]

    return prob

# Predict class of a sentence

def predict(sentence):
    prob_statement = classCondProb(sentence, 'statement') * prior('statement')
    prob_question = classCondProb(sentence, 'question') * prior('question')
    if  prob_statement > prob_question:
        return 'statement'
    else:
        return 'question'
```

In-lab exercise: Laplace smoothing

Run the code below and figure out why it fails.

When a word does not appear with a specific class in the training data, its class-conditional probability is 0, and we are unable to get a reasonable probability for that class.

Research Laplace smoothing, and modify the code above to implement Laplace smoothing (setting the frequency of all words with frequency 0 to a frequency of 1). Run the modified code on the test set.

In [15]:

```
test_docs = list([test['sentence'] for index,test in testing_data.iterrows()])
print('Getting prediction for "%s"' % test_docs[0])
predict(test_docs[0])
```

```
Getting prediction for "this is the book"

---------------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
<ipython-input-15-8095ea998923> in <module>
      1 test_docs = list([test['sentence'] for index,test in testing_data.iterrows()])
      2 print('Getting prediction for "%s"' % test_docs[0])
----> 3 predict(test_docs[0])

<ipython-input-14-1e60a1384353> in predict(sentence)
     29
     30 def predict(sentence):
---> 31     prob_statement = classCondProb(sentence, 'statement') * prior('statement')
     32     prob_question = classCondProb(sentence, 'question') * prior('question')
     33     if  prob_statement > prob_question:

<ipython-input-14-1e60a1384353> in classCondProb(sentence, className)
     19         if className == 'statement':
     20             idx = np.where(word_list_s == word)
---> 21             prob = prob * prob_s[np.array(idx)[0,0]]
     22         else:
     23             idx = np.where(word_list_q == word)

IndexError: index 0 is out of bounds for axis 1 with size 0
```

Exercise 1.1 (10 points)

Explain Why it failed and explain how to solve the problem.

Explanation here! (Double click to explain)

The failing has occured because we do not have the input word in our dictionary, the dictionary that we made using the training datasets for both of the classes separately. So the function idx = np.where(word_list_s == word)[0][0] is generating out of bound index error, since np.where(word_list_s == word)[0] returned empty list indexing empty array generated that error.\ So this is due to the absence of word in the training dataset for that class. What I did is just checked if the word is present in the word_list_s for statement and word_list_q for question condition. If the word is not present then I skipped that word by setting it's probability to 1 (Laplace smooting) and multiplied with the previous resulting probability, like this: prob = prob * 1.

Exercise 1.2 (20 points)

Modify your code and make it works.

In [16]:

```
# YOUR CODE HERE
#raise NotImplementedError()

# YOUR CODE HERE
def prior(className):
    denominator = len(stmt_docs) + len(question_docs)

    if className == 'statement':
        numerator =  len(stmt_docs)
    else:
        numerator =  len(question_docs)

    return np.divide(numerator,denominator)

# Calculate class conditional probability for a sentence

def classCondProb(sentence, className):
    words = get_words(sentence)
    prob = 1
    for word in words:
        if className == 'statement':
            if word in word_list_s:
                idx = np.where(word_list_s == word)
                prob = prob * prob_s[np.array(idx)[0,0]]
            else:
                prob = prob * 1 #laplace smoothing
        else:
            if word in word_list_q:
                idx = np.where(word_list_q == word)
                prob = prob * prob_q[np.array(idx)[0,0]]
            else:
                prob = prob * 1 #laplace smoothing

    return prob

# Predict class of a sentence

def predict(sentence):
    prob_statement = classCondProb(sentence, 'statement') * prior('statement')
    prob_question = classCondProb(sentence, 'question') * prior('question')
    if  prob_statement > prob_question:
        return 'statement'
    else:
        return 'question'
```

In [17]:

```
# Test function: Do not remove
test_docs = list([test['sentence'] for index,test in testing_data.iterrows()])

for sentence in test_docs:
    print('Getting prediction for %s"' % sentence)
    print(predict(sentence))

print("success!")
# End Test function
```

```
Getting prediction for this is the book"
question
Getting prediction for who are the novels characters"
question
Getting prediction for is this the author"
question
Getting prediction for I like apples"
question
success!
```

Expect result:\ Getting prediction for this is the book"\ question\ Getting prediction for who are the novels characters"\ question\ Getting prediction for is this the author"\ question

Take home exercise

Find a more substantial text classification dataset, clean up the documents, and build your NB classifier. Write a brief report on your in-lab and take home exercises and results.

In [18]:

```
import pandas as pd
```

Datasets

Dataset source: [http://www.cs.cornell.edu/people/pabo/movie-review-data (http://www.cs.cornell.edu/people/pabo/movie-review-data)](http://www.cs.cornell.edu/people/pabo/movie-review-data) This dataset includes two text files, one with positive movie reviews and the other with negative movie reviews. The total number of reviews on both text files is 5331, with one review per line. I'm going to create a Classifier using the Naive Bayes Classifier to determine whether the given review is positive or negative. Because the number of reviews on both classes is the same, I'm going to divide the training and test sets from both classes equally, 20% (almost 1000 reviews per class) for testing and 80% for training.

In [20]:

```
data_pos = pd.read_fwf('positive.txt')
data_neg = pd.read_fwf('negative.txt')

pdata = data_pos['Positive_text']
ndata = data_neg['Negative_text']

print("Number of positive Reviews", pdata.shape[0])
print("Number of negative Reviews", ndata.shape[0])
```

```
Number of positive Reviews 5331
Number of negative Reviews 5331
```

In [21]:

```
import random

random.seed(32)
idx = [i for i in range(pdata.shape[0])]
random.shuffle(idx)
train_len = int(0.8*len(idx))
train_idx = idx[:train_len]
test_idx = idx[train_len:]
```

In [22]:

```
ptrain = pdata.iloc[train_idx]
ptest = pdata.iloc[test_idx]
ntrain = ndata.iloc[train_idx]
ntest = ndata.iloc[test_idx]
```

In [23]:

```
print(ptrain.shape, ptest.shape, ntrain.shape, ntest.shape)
```

```
(4264,) (1067,) (4264,) (1067,)
```

In [ ]:

```

```

In [24]:

```
#outputs a series containing words as index with value as count
#input is a series of lines
def process_text(data):
    dp_split = data.str.split()
    dppd = pd.DataFrame(dp_split.to_list())
    dppd = dppd.fillna(value='bishal')

    sums = dppd[dppd[0].str.isalpha()][0].value_counts()
    for i in dppd.columns:
        if i!=0:
            sums = (dppd[dppd[i].str.isalpha()][i].value_counts()).radd(sums, fill_value=0)

    labels = []
    count = 0
    for word in sums.index:
        if len(word)<3 and word!="no":
            labels.append(word)

    labels.append('bishal')
    sums.drop(labels = labels, inplace=True)
    return sums
```

In [25]:

```
#bag of words for pos and neg class with frequecy
pos = process_text(ptrain)
neg = process_text(ntrain)

print(f"Number of words in positive class bag {pos.shape} \nNumber of words in negative class bag: {neg.shape}")

#bag of words in both postive class bag and negative class bag
tot = pos.add(neg, fill_value=0, axis = 0)
print("total number of words mixing both classes bags:", tot.shape)
```

```
Number of words in positive class bag (10372,)
Number of words in negative class bag: (10598,)
total number of words mixing both classes bags: (15297,)
```

In [26]:

```
# pos/tot[pos.index]
tot.head(10)
```

Out[26]:

```
aaa          1.0
aaliyah      3.0
aan          1.0
abandon      8.0
abandone     1.0
abandoned    2.0
abandono     1.0
abbas        1.0
abbass       1.0
abbott       2.0
dtype: float64
```

Preprocessing

The dataset is in text format and is well formatted, with each review on its own line. As a result, it will be simpler to divide the dataset into train and test sets. And, because the positive and negative reviews are in separate files, I can load them separately with a simple file loader.

Though the dataset is well formatted, the reviews themselves are not so clean, implying that the reviewer has given reviews in their own unique manner. Some reviews are in different languages, with non-english or near-english symbols (possibly a French character, I'm not sure:)). There are also other numeric characters (0-9) and symbols such as the dollar sign, at sign, comma, single quote, double quote, dash, tilde, and a few others. It would be simple to filter this out if they were written separately, but that was not the case. They are written within a text. Another major issue is the use of word with an apostrophe sign in between the characters in word. Using a simple strip method, the symbols on the outside of the word can be removed. However, I was unable to process the text that contained symbols within characters.

Another task that we can perform is word stemming, which is the process of replacing word variants with the base word. For example, the word eat can take several forms, including eaten, ate, eats, and eating. This example may not be useful in distinguishing between positive and negative reviews, but I hope it suffices. It would be preferable if we could simply do that.

So, what exactly did I do? I simply checked whether the word contained any characters other than the alphabets and filtered out any words that did. This is the most basic form of text processing, as well as the worst, because it may replace many important words (the words that serve the purpose to distinguish between positive and negative reviews). It turned out that there weren't many of these words. The total number of words in each class bag is around 10,000, and the total number of words in the collection of these bags is around 15,000. As a result, I was confident that this would assist me in developing a decently performing model.

I removed the word 'the' which has count around 5000, almost same as number of reviews, and some other two letter words (not-so-important word and some of them are not even a word) except some important one from the training datasets.

I didn't use any fancy text processing library. I just used the pandas to load the dataset and process the texts and store the bag of words and almost everything.

Model

In [27]:

```
#since there is equal number of positive and negative review in our training dataset the priors will be
#equal to 0.5
prob_y_pos = ptrain.shape[0]/(ntrain.shape[0]+ptrain.shape[0])
prob_y_neg = ntrain.shape[0]/(ntrain.shape[0]+ptrain.shape[0])

#probability of each word in its class
prob_word_pos = pos/tot[pos.index]
prob_word_neg = neg/tot[neg.index]
```

The model contains parameters like priors of each class, which in my case it's 0.5, since the number of data for each class is the same. And the other parameters are the probability of texts for each class. The calculations are made easy by the pandas dataframe and series data structures. Like wise during testing, the accessing of the probability of each words from the collection of bag of words is also made easy by the pandas series data structure.

In [28]:

```
#code to process the text sentence
def process_textsent(dppd, index): #dataframe of text, index to process the text at index
    return dppd.iloc[index][dppd.iloc[index].str.isalpha()]
```

In [29]:

```
def sentence_p_n_prob(sentence, prob_word_pos, prob_word_neg, prob_y_neg, prob_y_pos):
    pi_p = 1
    pi_n = 1
    for word in sentence.values:
        if word in prob_word_neg:
            pi_n = pi_n * prob_word_neg[word]
        if word in prob_word_pos:
            pi_p = pi_p * prob_word_pos[word]

    prob_s_n = prob_y_neg * pi_n
    prob_s_p = prob_y_pos * pi_p

    return prob_s_p, prob_s_n
```

In [30]:

```
def process_testdb(df):
    dp_split = df.str.split()
    dppd = pd.DataFrame(dp_split.to_list())
    dppd = dppd.fillna(value='@') #so that I could remove this while processing the text
    return dppd
```

In [31]:

```
processed_ptest = process_testdb(ptest)
processed_ntest = process_testdb(ntest)

#testing for positive reviews
sum_predict_p = 0
for index in range(processed_ptest.shape[0]):
    sentence = process_textsent(processed_ptest, index);
    prob_p, prob_n = sentence_p_n_prob(sentence, prob_word_pos, prob_word_neg, prob_y_neg, prob_y_pos)
    if prob_p >= prob_n:
        sum_predict_p += 1

#testing for negative reviews
sum_predict_n = 0
for index in range(processed_ntest.shape[0]):
    sentence = process_textsent(processed_ntest, index);
    prob_p, prob_n = sentence_p_n_prob(sentence, prob_word_pos, prob_word_neg, prob_y_neg, prob_y_pos)
    if prob_n >= prob_p:
        sum_predict_n += 1
```

In [33]:

```
print(f"Score for positive class: {sum_predict_p}/{ptest.shape[0]} \nScore for negative class: {sum_predict_n}/{ntest.shape[0]}")
```

```
Score for positive class: 803/1067
Score for negative class: 788/1067
```

In [34]:

```
pos_per = sum_predict_p*100/ptest.shape[0]
neg_per = sum_predict_n*100/ntest.shape[0]
print("Percentage of correct postive reviews: ", pos_per)
print("Percentage of correct negative reviews: ", neg_per)
print("Total accuracy of model: ", (pos_per + neg_per) / 2)
```

```
Percentage of correct postive reviews:  75.25773195876289
Percentage of correct negative reviews:  73.8519212746017
Total accuracy of model:  74.55482661668229
```

Result

My model's scores for each class are listed below: \ Positive class score: 803/1067 score for negative class: 788/1067 My model correctly classifies 803 of the 1067 reviews for the positive class and 788 for the negative class. The percentage scores are as follows: 75.25773195876289% of correct positive reviews 73.8519212746017% of correct negative reviews So, based on 2000 reviews (1000 for each class), the overall accuracy of my model is 74.44%. The performance can be improved by properly preprocessing text datasets and using a large number of such training datasets. No words should be filtered out solely based on their character composition. Proper word stemming may also result in improved model performance.

In [ ]:

```
```