Raul G. Martinez (PID: A12461871)

DSE 210: Probability and statistics Winter 2020

Worksheet 6 — Generative models 2

1. Would you expect the following pairs of random variables to be uncorrelated, positively correlated, or negatively correlated?

(a) The weight of a new car and its price.

Uncorrelated

(b) The weight of a car and the number of seats in it.

Positively Correlated

(c) The age in years of a second-hand car and its current market value.

Negatively Correlated

2. Consider a population of married couples in which every wife is exactly 0.9 of her husband's age. What is the correlation between husband's age and wife's age?

Because every wife is exactly 0.9 of her husband's age there is **perfect positive correlation**, therefore if we make a 2-D scatter plot **r = 1**

3. Each of the following scenarios describes a joint distribution (x, y). In each case, give the parameters of the (unique) bivariate Gaussian that satisfies these properties.

(a) x has mean 2 and standard deviation 1, y has mean 2 and standard deviation 0.5, and the correlation between x and y is -0.5.

Given properties: $\mu_x = 2\,, \mu_y = 2\,, \sigma_x = 1\,, \sigma_y = 0.5, corr(x,y) = -0.5$

The parameters for the (unique) bivariate Gaussian are:

$cov(x,y) = \sigma_x\sigma_y\ corr(x,y) = (1) * (0.5) * (-0.5) = -0.25$

$\Sigma_{xx} = var(x) = \sigma_x{}^2 = (1)^2 = 1$

$\Sigma_{yy} = var(y) = \sigma_y{}^2 = (0.5)^2 = 0.25$

$\Sigma_{xy} = \Sigma_{yx} = cov(x,y) = -0.25$

Therefore, the covariance matrix becomes:

$$\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma_{xx}} & \boldsymbol{\Sigma_{xy}} \\ \boldsymbol{\Sigma_{yx}} & \boldsymbol{\Sigma_{yy}} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & \mathbf{-0.25} \\ \mathbf{-0.25} & \mathbf{0.25} \end{bmatrix}$$

(b) x has mean 1 and standard deviation 1, and y is equal to x.

Given properties: $\mu_x = 1$, $\mu_y = \mu_x$, $\sigma_x = 1$, $\sigma_y = \sigma_x$, $corr(x,y) = 1$

The parameters for the (unique) bivariate Gaussian are:

$cov(x,y) = \sigma_x\sigma_y\ corr(x,y) = (1) * (1) * (1) = 1$

$\Sigma_{xx} = var(x) = {\sigma_x}^2 = (1)^2 = 1$

$\Sigma_{yy} = var(y) = {\sigma_y}^2 = (1)^2 = 1$

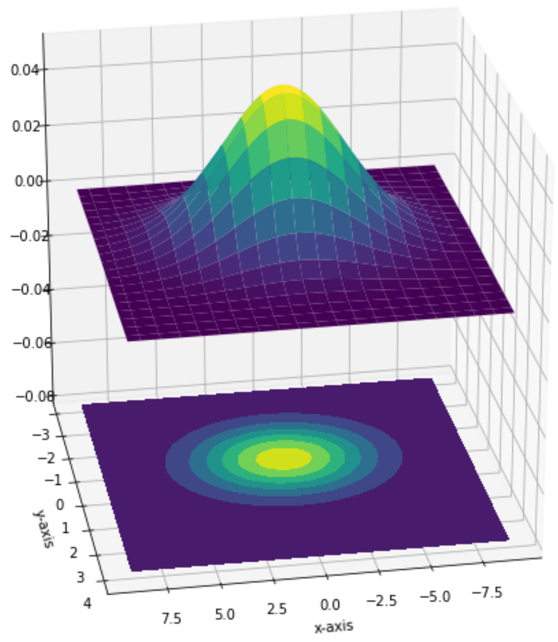
$\Sigma_{xy} = \Sigma_{yx} = cov(x,y) = 1$

Therefore, the covariance matrix becomes:

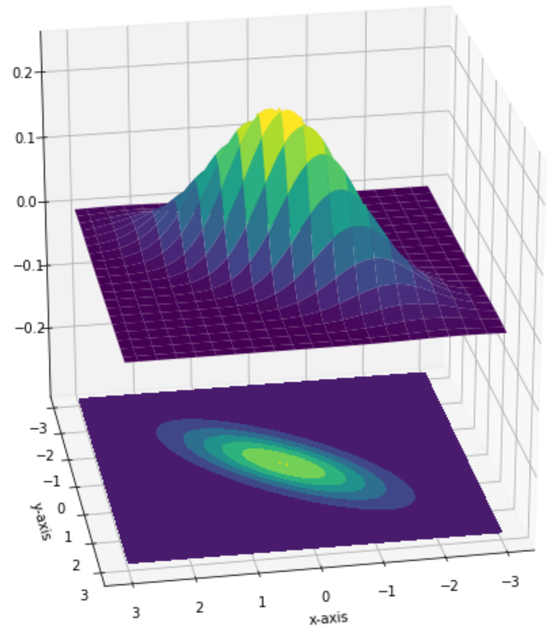
$$\boldsymbol{\Sigma} = \begin{bmatrix} \boldsymbol{\Sigma_{xx}} & \boldsymbol{\Sigma_{xy}} \\ \boldsymbol{\Sigma_{yx}} & \boldsymbol{\Sigma_{yy}} \end{bmatrix} = \begin{bmatrix} \mathbf{1} & \mathbf{1} \\ \mathbf{1} & \mathbf{1} \end{bmatrix}$$

4. Roughly sketch the shapes of the following Gaussians N(μ, Σ). For each, you only need to show a representative contour line which is qualitatively accurate (has approximately the right orientation, for instance).

(a) $\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ *and* $\Sigma = \begin{bmatrix} 9 & 0 \\ 0 & 1 \end{bmatrix}$



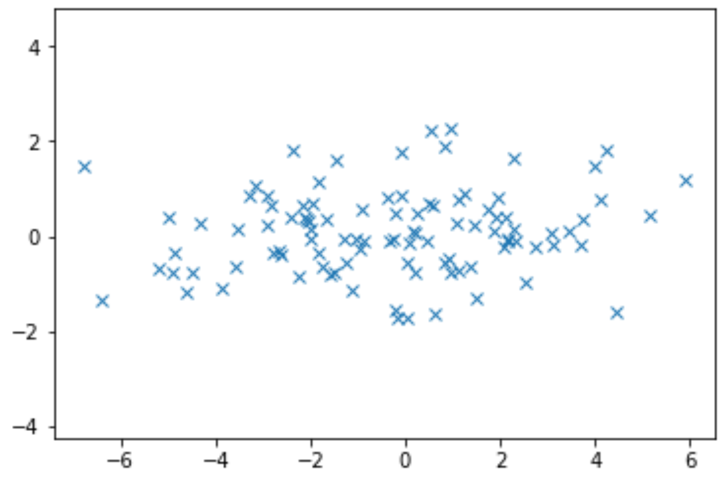
(b) $\mu = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$ *and* $\Sigma = \begin{bmatrix} 1 & -0.75 \\ -0.75 & 1 \end{bmatrix}$



5. For each of the two Gaussians in the previous problem, check your answer using Python: draw 100 random samples from that Gaussian and plot it.

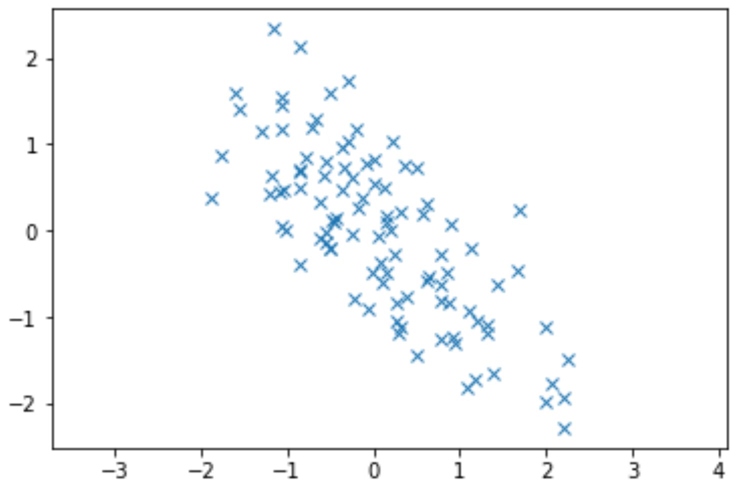
Part 4 (a):

```
import matplotlib.pyplot as plt
import numpy as np
Sigma = np.array([[ 9 , 0], [0,  1]])
x, y = np.random.multivariate_normal([0,0], Sigma, 100).T
plt.plot(x, y, 'x')
plt.axis('equal')
plt.show()
```



Part 4 (b):

```
import matplotlib.pyplot as plt
import numpy as np
Sigma = np.array([[ 1 , -0.75], [-0.75,  1]])
x, y = np.random.multivariate_normal([0,0], Sigma, 100).T
plt.plot(x, y, 'x')
plt.axis('equal')
plt.show()
```



DSE 210: Probability and statistics Winter 2020

Worksheet 8 — Generative models 3

6. Consider the linear classifier $w \cdot x \geq \theta$, where

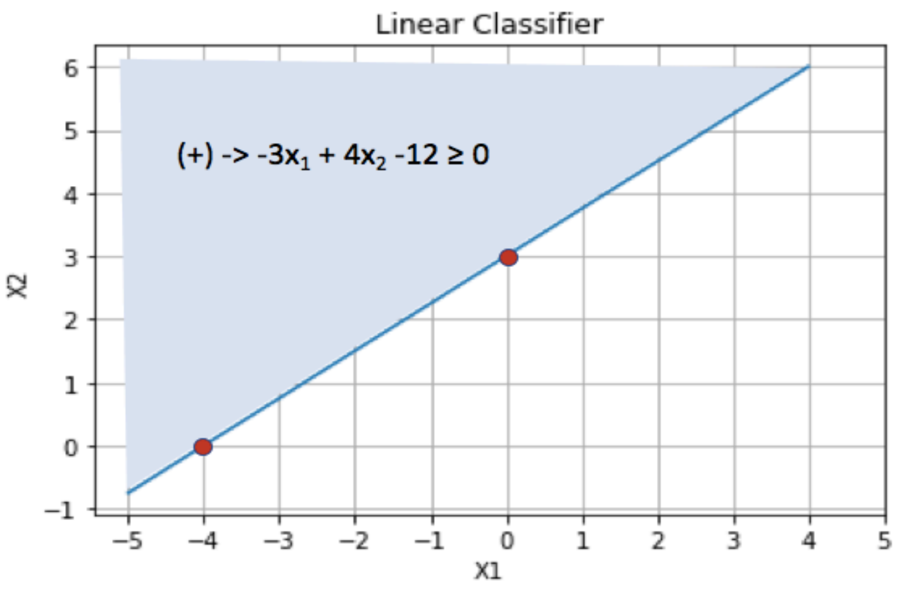
$$w = \begin{bmatrix} -3 \\ 4 \end{bmatrix} \text{ and } \theta = 12$$

Sketch the decision boundary in R^2. Make sure to label precisely where the boundary intersects the coordinate axes, and also indicate which side of the boundary is the positive side.

The linear classifier above can be expressed as:

$$-3x_1 + 4x_2 \geq 12$$

The plot below shows the decision boundary (blue line) for the linear classifier. The red dots indicate where the boundary intersects the axes x1 and x2, and the shaded light-blue region indicates the positive side boundary given by the expression showed inside such region. Additionally, the negative region can be expressed as $-3x_1 + 4x_2 - 12 < 0$



7. How many parameters are needed to specify a diagonal Gaussian in R^d?

We need to specify the mean and standard deviation for each dimension, therefore for a diagonal Gaussian in R^d we need:

$$\mu_1, \mu_2, \mu_3, \ldots, \mu_d \quad \text{and} \quad \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_d$$

Then, we can define the covariance matrix as (with all off-diagonal elements as zero):

$$\Sigma = diag({\sigma_1}^2, {\sigma_2}^2, {\sigma_3}^2, \ldots, {\sigma_d}^2)$$

8. Text classification using multinomial Naive Bayes.

(a) For this problem, you'll be using the 20 Newsgroups data set. There are several versions of it on the web. You should download "20news-bydate.tar.gz" from

http://qwone.com/~jason/20Newsgroups/

Unpack it and look through the directories at some of the files. Overall, there are roughly 19,000 documents, each from one of 20 newsgroups. The label of a document is the identity of its newsgroup. The documents are divided into a training set and a test set.

(b) The same website has a processed version of the data, "20news-bydate-matlab.tgz", that is particularly convenient to use. Download this and also the file "vocabulary.txt". Look at the first training document in the processed set and the corresponding original text document to under- stand the relation between the two.

(c) The words in the documents constitute an overall vocabulary V of size 61188. Build a multinomial Naive Bayes model using the training data. For each of the 20 classes j = 1, 2, . . . , 20, you must have the following:

- *πj, the fraction of documents that belong to that class; and*
- *Pj, a probability distribution over V that models the documents of that class.*

In order to fit P_j, imagine that all the documents of class j are strung together. For each word w ⍰ V , let P_{jw} be the fraction of this concatenated document occupied by w. Well, almost: you will need to do smoothing (just add one to the count of how often w occurs).

8. Text Classification Using Multinomial Naive Bayes

In [1]:

```
# import libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
import nltk
from nltk.corpus import stopwords
```

(c) Calculate Pi, the fraction of documents that belong to each class j

In [2]:
```
# define function to calculate pi
def fraction_doc_classj(labels):
    """
    Find the class probabilities pi for each class j
    """

    # define dictionary for pi_j where j is each class 1,2,..., 20
    pi = {}
    No_Classes = 20
    for j in range(1,No_Classes+1):
        pi[j] = 0

    # count occurrences for a dataframe type variable
    if isinstance(labels, pd.core.series.Series):
        lines = list(labels)
    else:
        lines = labels.readlines()

    # count the occurrence of each class j
    for line in lines:
        if isinstance(labels, pd.core.series.Series):
            j_val = line
        else:
            j_val = int(line.split()[0])
        pi[j_val] += 1

    # divide each class count for the total number of documents
    for j in pi.keys():
        pi[j] /= len(lines)
    return pi
```

In [3]:
```
# read labels for train data
train_label = open('/Users/gio/Documents/DSE/2019-rgm001/DSE210/Lecture4/20news-bydate/matlab/train.label', 'r')

pi = fraction_doc_classj(train_label)
print(pi)
```

```
{1: 0.04259472890229834, 2: 0.05155736977549028, 3: 0.05075871860857219, 4: 0.05208980388676901, 5: 0.051024935664211
554, 6: 0.052533498979501284, 7: 0.051646108794036735, 8: 0.052533498979501284, 9: 0.052888455053687104, 10: 0.052710
9770165942, 11: 0.05306593309078002, 12: 0.0527109770165942, 13: 0.05244475996095483, 14: 0.0527109770165942, 15: 0.0
52622237998047744, 16: 0.05315467210932647, 17: 0.04836276510781791, 18: 0.05004880646020055, 19: 0.0411749046055550
6, 20: 0.03336587097346703 5}
```

(c) Create dataframe for training data and training labels

In [4]:
```
# create function to merge both
def merge_data_labels_df(data, labels):
    """
    Create dataframe by increasing size of labels to match data
    """
    # get array for documents and class indexes
    docIdx = train_data['docIdx']
    classIdx = train_labels['classIdx']

    # match data and label size by increasing label length
    new_train_labels = []
    i = 0
    for idx in range(len(docIdx)-1):
        new_train_labels.append(classIdx[i])
        if docIdx[idx] != docIdx[idx+1]:
            i += 1
    new_train_labels.append(classIdx[i])

    # create dataframe with both train and label
    df = train_data
    df['classIdx'] = new_train_labels

    return df
```

In [5]:
```
# read train data and labels
train_data = pd.read_csv('/Users/gio/Documents/DSE/2019-rgm001/DSE210/Lecture4/20news-bydate/matlab/train.data',
                         delimiter=' ', names=['docIdx', 'wordIdx', 'count'])
train_labels = pd.read_csv('/Users/gio/Documents/DSE/2019-rgm001/DSE210/Lecture4/20news-bydate/matlab/train.label',
                           names=['classIdx'])

df = merge_data_labels_df(train_data,train_labels)

df.head()
```

Out[5]:

	docIdx	wordIdx	count	classIdx
0	1	1	4	1
1	1	2	2	1
2	1	3	10	1
3	1	4	4	1
4	1	5	2	1

(c) Use Laplace Smoothing to calculate the probability of each word per class, Pjw

```
In [6]: def Prob_word_per_class(df, remove_stopwords, stopwordsIdx):
            """
            Find Pjw and apply Laplace smoothing with alpha parameter
            """

            # remove stopwords if wanted
            if remove_stopwords:
                df = df[~df.wordIdx.isin(idx)]

            # parameters
            alpha = 0.001
            No_class = 20

            # calculate the fraction of the concatenated doccuments occupied by w
            word_wj = df.groupby(['classIdx','wordIdx'])
            word_j = df.groupby(['classIdx'])
            Pr_jw =  (word_wj['count'].sum() + alpha) / (word_j['count'].sum() + 61188 + 1)
            Pr_jw = Pr_jw.unstack()

            # replace missing values with the constant alpha/(count+|V|+1)
            # where 'count' is how often w occurs and |V| is the size of the vocabulary
            for j in range(1,No_class+1):
                Pr_jw.loc[j,:] = Pr_jw.loc[j,:].fillna(alpha/(word_j['count'].sum()[j] + 61188 + 1))

            return Pr_jw.to_dict()
```

```
In [7]: Pr_jw = Prob_word_per_class(df, remove_stopwords=False, stopwordsIdx = [])
        len(Pr_jw)

Out[7]: 53975
```

(d) Write a routine that uses this naive Bayes model to classify a new document. To avoid underflow, work with logs rather than multiplying together probabilities.

(d) Routine with Naive Bayes to Classify a New Document

```
In [8]: def Multinomial_NB(df, pi, Pr_jw, log_replacement = False):
            """
            Model Function for Multinomial Naive Bayes

            Inputs:
            - df: columns = ['docIdx', 'wordIdx', 'count']

            Model Equation (in Log scale):
            armax_j  log(pi_j) + Sigma Sum (from w = 1 to |V|) X_i * log (Pr_jw)

            where,
            - pi_j = fraction of documents that belong to that class j
            - Pr_jw = fraction of each word for a given class
            - j = class
            - w = each word from vocabulary
            - |V| = lenght of vocabulary
            """

            # convert dataframe to dict with format for faster speed
            df_dict = df.to_dict()
            data_dict = {}
            for df_row in df_dict['docIdx'].keys():
                doc_index = df_dict['docIdx'][df_row]
                word_index = df_dict['wordIdx'][df_row]

                try:
                    data_dict[doc_index][word_index] = df_dict['count'][df_row]
                except:
                    data_dict[df_dict['docIdx'][df_row]] = {}
                    data_dict[doc_index][word_index] = df_dict['count'][df_row]

            # use equation to find the score and take arg max as the prediction
            predicted = []
            for doc in data_dict.keys(): # loop over every document
                score_dict = {}
                No_class = 20
                for class_ in range(1,No_class+1): # calculate a score for each class
                    score_dict[class_] = 1 # initialize score value
                    for word in data_dict[doc]: # loop over every word in the document

                        # calculate right-hand side (Sigma sum) of the model equation
                        try:
                            # calculate with log replacement log(1+f)
                            if log_replacement:
                                score_dict[class_] += np.log(1+data_dict[doc][word]) * np.log(Pr_jw[word][class_])
                            else: # else do just f
                                score_dict[class_] += data_dict[doc][word] * np.log(Pr_jw[word][class_])
                        except:
                            # missing words in the vocabulary yield a zero score, X_i = 0
                            score_dict[class_] += 0

                    # add left-hand side (log pi) of the model equation
                    score_dict[class_] +=  np.log(pi[class_])

                # get class with max probability in each document
                max_score = max(score_dict, key=score_dict.get)
                predicted.append(max_score)

            return predicted
```

(e) Evaluate the performance of your model on the test data. What error rate do you achieve?

(e) Evaluate the performance of the model with test data

```
In [9]: def predict_error(prediction, labels):
            """
            Find predicted error rate
            """
            correct = 0
            for i,j in zip(prediction, labels):
                if i == j:
                    correct +=1

            perc_error = 100*(1-(correct/len(labels)))

            return round(perc_error,6)
```

```
In [10]: test_data = pd.read_csv('/Users/gio/Documents/DSE/2019-rgm001/DSE210/Lecture4/20news-bydate/matlab/test.data',
                                delimiter=' ', names=['docIdx', 'wordIdx', 'count'])
         test_labels = pd.read_csv('/Users/gio/Documents/DSE/2019-rgm001/DSE210/Lecture4/20news-bydate/matlab/test.label',
                                  names=['classIdx'])

         test_labels = test_labels.classIdx.tolist()

         predict = Multinomial_NB(test_data, pi, Pr_jw, log_replacement = False)
```

```
In [11]: predict_error(predict, test_labels)

Out[11]: 22.238508
```

(f) If you have the time and inclination: see if you can get a better-performing model.

- *Split the training data into a smaller training set and a validation set. The split could be 80-20, for instance. You'll use this training set to estimate parameters and the validation set to decide between different options.*
- *Think of 2-3 ways in which you might improve your earlier model. Examples include: (i) replacing the frequency f of a word in a document by log(1 + f), (ii) removing stopwords; (iii) reducing the size of the vocabulary; etc. Estimate a revised model for each of these, and use the validation set to choose between them.*
- *Evaluate your final model on the test data. What error rate do you achieve?*

(f) split data into smaller training set and validation

```
In [12]: train_data, val_data, train_labels, val_labels = train_test_split(df[['docIdx','wordIdx','count']],
                                                                           df['classIdx'], test_size=0.2, random_state=1)
```

```
In [13]: # separate validation set labels to match predicted size
         new_val_labels = pd.DataFrame({'docIdx': val_data['docIdx'], 'classIdx':val_labels})
         new_val_labels.drop_duplicates(keep = 'first', inplace = True)
         new_val_labels = new_val_labels['classIdx']
```

(f) implement strategies to improve earlier model

a) Replacing the frequency f of a word in a document by log(1 + f)
b) Removing Stopwords

Different Models to evaluate with validation data

```
1) frequency f
2) frequency f and removing stop words
3) frequency log(1+f)
4) frequency log(1+f) and removing stop words
```

In [14]:
```
# import stopwords list from nltk libraries
# nltk.download()
stopwords_list = stopwords.words('english')

print(stopwords_list)
```

```
['i', 'me', 'my', 'myself', 'we', 'our', 'ours', 'ourselves', 'you', "you're", "you've", "you'll", "you'd", 'your',
'yours', 'yourself', 'yourselves', 'he', 'him', 'his', 'himself', 'she', "she's", 'her', 'hers', 'herself', 'it', "i
t's", 'its', 'itself', 'they', 'them', 'their', 'theirs', 'themselves', 'what', 'which', 'who', 'whom', 'this', 'tha
t', "that'll", 'these', 'those', 'am', 'is', 'are', 'was', 'were', 'be', 'been', 'being', 'have', 'has', 'had', 'havi
ng', 'do', 'does', 'did', 'doing', 'a', 'an', 'the', 'and', 'but', 'if', 'or', 'because', 'as', 'until', 'while', 'o
f', 'at', 'by', 'for', 'with', 'about', 'against', 'between', 'into', 'through', 'during', 'before', 'after', 'abov
e', 'below', 'to', 'from', 'up', 'down', 'in', 'out', 'on', 'off', 'over', 'under', 'again', 'further', 'then', 'onc
e', 'here', 'there', 'when', 'where', 'why', 'how', 'all', 'any', 'both', 'each', 'few', 'more', 'most', 'other', 'so
me', 'such', 'no', 'nor', 'not', 'only', 'own', 'same', 'so', 'than', 'too', 'very', 's', 't', 'can', 'will', 'just',
'don', "don't", 'should', "should've", 'now', 'd', 'll', 'm', 'o', 're', 've', 'y', 'ain', 'aren', "aren't", 'could
n', "couldn't", 'didn', "didn't", 'doesn', "doesn't", 'hadn', "hadn't", 'hasn', "hasn't", 'haven', "haven't", 'isn',
"isn't", 'ma', 'mightn', "mightn't", 'mustn', "mustn't", 'needn', "needn't", 'shan', "shan't", 'shouldn', "should
n't", 'wasn', "wasn't", 'weren', "weren't", 'won', "won't", 'wouldn', "wouldn't"]
```

In [15]:
```
# function to get the stop word indexes from vocabulary
def get_stopwords_idx(stopwords_list):
    """
    Use vocabulary .txt file from 20 Newsgroups data set
    """

    # read current vocabulary
    V = pd.read_csv('/Users/gio/Documents/DSE/2019-rgm001/DSE210/Lecture4/vocabulary.txt', names=['word'])

    # find stopwords index from the vocabulary
    stopwords_idx = V[V.word.isin(stopwords_list)].index

    return stopwords_idx
```

In [16]:
```
# Train the model
# define pi
pi = fraction_doc_classj(train_labels)

# define dataframe for train data
df_train = train_data
df_train['classIdx'] = train_labels

# define probability of each word per class
Pr_jw = Prob_word_per_class(df_train, remove_stopwords=False, stopwordsIdx = [])
idx = get_stopwords_idx(stopwords_list)
Pr_jw_wo_stopwords = Prob_word_per_class(df_train, remove_stopwords=True, stopwordsIdx = idx)
```

In [17]:
```
# use validation dataset on all 4 models listed above

predict_1 = Multinomial_NB(val_data, pi, Pr_jw, log_replacement = False)
predict_2 = Multinomial_NB(val_data, pi, Pr_jw_wo_stopwords, log_replacement = False)
predict_3 = Multinomial_NB(val_data, pi, Pr_jw, log_replacement = True)
predict_4 = Multinomial_NB(val_data, pi, Pr_jw_wo_stopwords, log_replacement = True)
```

```
In [18]: # print the error for each of the 4 model options above
         models = ['frequency f', 'frequency f and removing stop words', 'frequency log(1+f)',
                   'frequency log(1+f) and removing stop words']
         predictions = [predict_1, predict_2, predict_3, predict_4]

         i = 1
         for m, p in zip(models, predictions):
             error = predict_error(p, list(new_val_labels))
             print(str(i) + ') ' + str(m) + '\nError Rate = ' + str(error) + '\n')
             i += 1
```

```
1) frequency f
Error Rate = 28.425358

2) frequency f and removing stop words
Error Rate = 28.318663

3) frequency log(1+f)
Error Rate = 28.051925

4) frequency log(1+f) and removing stop words
Error Rate = 28.185294
```

(f) Evaluate the final model on the test data

```
In [19]: # use model #3 above "frequenct log(1+f)" as it gave the best error rate with the validation set
         final_predict = Multinomial_NB(test_data, pi, Pr_jw, log_replacement = True)
```

```
In [20]: # calculate error for final model
         predict_error(final_predict, test_labels)
```

```
Out[20]: 22.918055
```

9. Handwritten digit recognition using a Gaussian generative model. In class, we mentioned the MNIST data set of handwritten digits. You can obtain it from:

http://yann.lecun.com/exdb/mnist/index.html

In this problem, you will build a classifier for this data, by modeling each class as a multivariate (784-dimensional) Gaussian.

(a) Upon downloading the data, you should have two training files (one with images, one with labels) and two test files. Unzip them.
In order to load the data into Python you will find the following code helpful:

http://cseweb.ucsd.edu/~dasgupta/dse210/loader.py

For instance, to load in the training data, you can use:

x,y = loadmnist('train-images-idx3-ubyte', 'train-labels-idx1-ubyte')

This will set x to a 60000 x 784 array where each row corresponds to an image, and y to a length-60000 array where each entry is a label (0-9). There is also a routine to display images: use displaychar(x[0]) to show the first data point, for instance.

9. Handwritten digit recognition using a Gaussian generative model

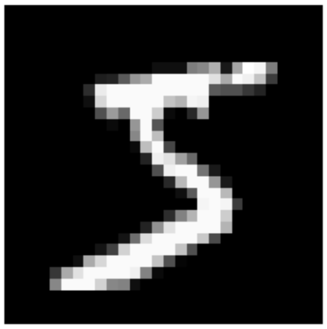
In [1]:
```
# import libraries
from struct import unpack
import numpy as np
import matplotlib.pylab as plt
import gzip
from sklearn.model_selection import train_test_split
from collections import Counter
from scipy.stats import multivariate_normal
import random
```

(a) Load the MNIST training data.

In [2]:
```
# define load and display functions
def loadmnist(imagefile, labelfile):
    """
    Load the MNIST data set
    """
    # Open the images with gzip in read binary mode
    images = gzip.open(imagefile, 'rb')
    labels = gzip.open(labelfile, 'rb')

    # Get metadata for images
    images.read(4)  # skip the magic_number
    number_of_images = images.read(4)
    number_of_images = unpack('>I', number_of_images)[0]
    rows = images.read(4)
    rows = unpack('>I', rows)[0]
    cols = images.read(4)
    cols = unpack('>I', cols)[0]

    # Get metadata for labels
    labels.read(4)
    N = labels.read(4)
    N = unpack('>I', N)[0]

    # Get data
    x = np.zeros((N, rows*cols), dtype=np.uint64)  # Initialize numpy array
    y = np.zeros(N, dtype=np.uint8)  # Initialize numpy array
    for i in range(N):
        for j in range(rows*cols):
            tmp_pixel = images.read(1)  # Just a single byte
            tmp_pixel = unpack('>B', tmp_pixel)[0]
            x[i][j] = tmp_pixel
        tmp_label = labels.read(1)
        y[i] = unpack('>B', tmp_label)[0]

    images.close()
    labels.close()
    return (x, y)

def displaychar(image):
    plt.imshow(np.reshape(image, (28,28)), cmap=plt.cm.gray)
    plt.axis('off')
    plt.show()
```

```
In [3]: # read image and label .gz files
        imagefile_train = '/Users/gio/Documents/DSE/2019-rgm001/DSE210/Lecture4/train-images-idx3-ubyte.gz'
        labelfile_train = '/Users/gio/Documents/DSE/2019-rgm001/DSE210/Lecture4/train-labels-idx1-ubyte.gz'
        x_train,y_train = loadmnist(imagefile_train, labelfile_train)

        print(x_train.shape)
        print(y_train.shape)

        (60000, 784)
        (60000,)
```

```
In [4]: # show first data point
        displaychar(x_train[0])
```



(b) Split the training set into two pieces – a training set of size 50000, and a separate validation set of size 10000. Also load in the test data.

(b) Split the training set into training (size 50000) and validation (size 10000). Also load the test data.

```
In [5]: # split into train and validation
        x_train, x_val, y_train, y_val = train_test_split(x_train,y_train, test_size=1/6, random_state=1)

        imagefile_test = '/Users/gio/Documents/DSE/2019-rgm001/DSE210/Lecture4/t10k-images-idx3-ubyte.gz'
        labelfile_test = '/Users/gio/Documents/DSE/2019-rgm001/DSE210/Lecture4/t10k-labels-idx1-ubyte.gz'

        print(x_train.shape)
        print(y_train.shape)
        print(x_val.shape)
        print(y_val.shape)

        (50000, 784)
        (50000,)
        (10000, 784)
        (10000,)
```

```
In [6]: # load test data
        x_test,y_test = loadmnist(imagefile_test, labelfile_test)

        print(x_test.shape)
        print(y_test.shape)

        (10000, 784)
        (10000,)
```

(c) Now fit a Gaussian generative model to the training data of 50000 points:

- *Determine the class probabilities: what fraction ▯$_0$ of the training points are digit 0, for instance? Call these values $\pi_0, \ldots, \pi_9$.*
- *Fit a Gaussian to each digit, by finding the mean and the covariance of the corresponding data points. Let the Gaussian for the jth digit be $P_j = N(\mu_j, \Sigma_j)$.*

Using these two pieces of information, you can classify new images x using Bayes' rule: simply pick the digit j for which $\pi_j P_j(x)$ is largest.

(c) Fit a Gaussian generative model to the training data of 50000 pts.

In [7]:
```
# determine the class probabilities
pi = Counter(y_train)
for y, count in pi.items():
    pi[y] = count/len(y_train)
print(pi)
```

```
Counter({1: 0.11184, 7: 0.10422, 3: 0.10204, 9: 0.09966, 2: 0.0996, 0: 0.09942, 6: 0.0981, 8: 0.09782, 4: 0.0972, 5:
0.0901})
```

In [8]:
```
# generate dictionary with train data, labels as keys and data as lists
train_data_dict = {}
for idx, train_data in enumerate(x_train):
    curr_y = y_train[idx] # get current label
    if curr_y in train_data_dict:
        temp_data = train_data_dict[curr_y]
        temp_data.append(train_data)
    else:
        train_data_dict[curr_y] = [train_data]

train_data_dict.keys()
```

Out[8]:
```
dict_keys([8, 6, 3, 0, 2, 4, 1, 9, 7, 5])
```

In [9]:
```
# calculate mean and covariance
mean_dict, cov_dict = {}, {}
for y, x in train_data_dict.items():
    mean_dict[y] = np.mean(x, axis=0)
    cov_dict[y] = np.cov(x, rowvar=False)

print(mean_dict.keys())
print(cov_dict.keys())
```

```
dict_keys([8, 6, 3, 0, 2, 4, 1, 9, 7, 5])
dict_keys([8, 6, 3, 0, 2, 4, 1, 9, 7, 5])
```

In [10]:
```
# Fit a Gaussian to each digit, let it be Pj = N(mean_j, covariance_j)
def Gaussian(x, pi, mean_dict, cov_dict, c):
    """
    Fit a Gaussian to each digit
    Pj = N(mean_j, covariance_j)
    """
    Prob_gaussian = []
    for j in range(len(train_data_dict)):
        # smooth covariance matrix with cI, where c is a constant and I is the identity matrix
        cov_smooth = cov_dict[j] + (c * np.identity(784))

        # work in log scale for pi_j*Pj for each j
        log_pi_j = np.log(pi[j])
        log_Pj = multivariate_normal.logpdf(x, mean_dict[j], cov_smooth)

        Prob_gaussian.append(log_pi_j + log_Pj)
    return Prob_gaussian
```

(d) One last step is needed: it is important to smooth the covariance matrices, and the usual way to do this is to add in cI, where c is some constant and I is the identity matrix. What value of c is right? Use the validation set to help you choose. That is, choose the value of c for which the resulting classifier makes the fewest mistakes on the validation set. What value of c did you get?

(d) What value of c is right? Use the validation set to choose.

```
In [22]: # function to find the error rate
def get_error_rate(x, y, pi, mean_dict, cov_dict, c):
    """
    Find error rate for predicted and label
    """
    count_wrong = 0
    missclass_digits = []
    for idx, data in enumerate(x):

        # get predicted label from gaussian model, pick the max digit j
        prob_gaussian = Gaussian(data, pi, mean_dict, cov_dict, c)
        predicted = prob_gaussian.index(max(prob_gaussian))

        # find number of wrong predictions
        if predicted != y[idx]:
            count_wrong += 1
            missclass_digits.append((predicted, data))

    # return error rate and missclassified digits
    error_rate = float(count_wrong)/len(x)
    return (error_rate, missclass_digits)
```

```
In [13]: # test different c values in the validation set
c_values = [0.1, 10, 100, 1000, 10000, 100000]
error_rate_list = []
for c in c_values:
    # only do 1000 values from the validation set
    error_rate, missclass_digits = get_error_rate(x_val[:1000], y_val[:1000], pi, mean_dict, cov_dict, c)
    error_rate_list.append(error_rate)
```

```
In [14]: # print c values and error rate
for c,e in zip(c_values, error_rate_list):
    print('c-value: ',c,'\t Error Rate: ', e)
```

```
c-value:  0.1    Error Rate:  0.204
c-value:  10     Error Rate:  0.135
c-value:  100    Error Rate:  0.096
c-value:  1000   Error Rate:  0.066
c-value:  10000          Error Rate:  0.06
c-value:  100000         Error Rate:  0.163
```

(e) Turn in an iPython notebook that includes:

- *All your code.*
- *Error rate on the MNIST test set.*
- *Out of the misclassified test digits, pick five at random and display them. For each instance, list the posterior probabilities Pr(y|x) of each of the ten classes.*

(e) Find the error rate on the test data, pick five at random to display, and list the posterior probailities Pr(y|x)

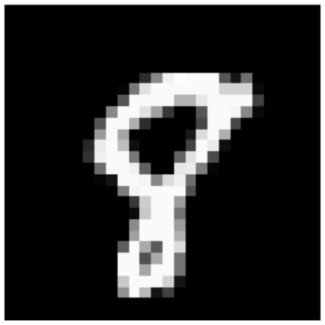
```
In [15]: # use found c value and model to find the error rate on the MNIST test set
         c = 10000
         error_rate_test, missclass_digits_test = get_error_rate(x_test, y_test, pi, mean_dict, cov_dict, c)
```

```
In [23]: print('MNIST Test Set, Error Rate: ', 100*error_rate_test, '%')
```

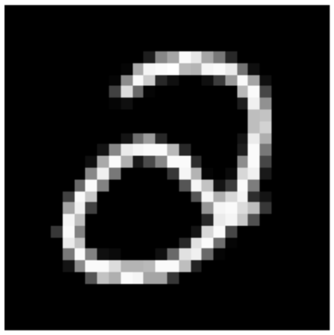
```
MNIST Test Set, Error Rate:  5.2 %
```

```
In [54]: # display 5 prediction errors at random
         five_digits = []
         digit = 1
         for y, data in random.choices(missclass_digits_test, k=5):
             five_digits.append(data)
             print('Digit', digit, '-> Predicted:', y, '; Data:')
             displaychar(data)
             digit += 1
```

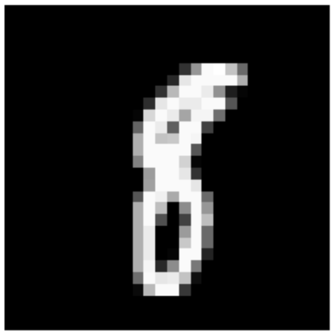
```
Digit 1 -> Predicted: 9 ; Data:
```



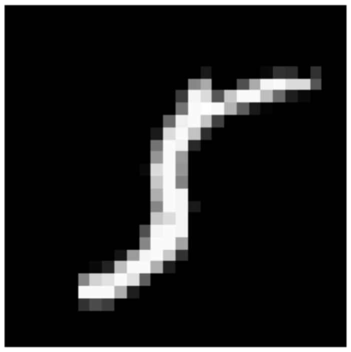
```
Digit 2 -> Predicted: 0 ; Data:
```



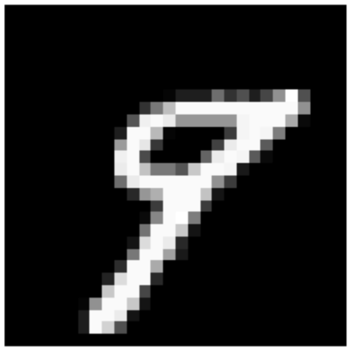
```
Digit 3 -> Predicted: 1 ; Data:
```



```
Digit 4 -> Predicted: 1 ; Data:
```



```
Digit 5 -> Predicted: 7 ; Data:
```



```
In [56]: # list the posterior probability Pr(y|x) of each of the ten classes
         """
         We can calculate the posterior probability as:
         log(Pr(y|x)) = log(Pr(x|y)) + log(Pr(y))

         The evidence term Pr(x) can be ignored by normalizing across the classes
         """

         # calculate the posterior probability for each class
         c = 10000
         log_Pr_y_x = Gaussian(five_digits, pi, mean_dict, cov_dict, c)

         # display the probabilities
         for j, data in enumerate(log_Pr_y_x):
             print('Label: ', j)
             for k, digit in enumerate(data):
                 print('Log Probability for digit', k+1, ': ', digit)
```

```
Label:  0
Log Probability for digit 1 :  -4483.697084518334
Log Probability for digit 2 :  -4444.30697586138
Log Probability for digit 3 :  -4465.044400905459
Log Probability for digit 4 :  -4457.331395261728
Log Probability for digit 5 :  -4503.799358574147
Label:  1
Log Probability for digit 1 :  -4480.386065294797
Log Probability for digit 2 :  -4546.324965009476
Log Probability for digit 3 :  -4403.689499990516
Log Probability for digit 4 :  -4409.697307134239
Log Probability for digit 5 :  -4482.298690276768
Label:  2
Log Probability for digit 1 :  -4467.34197460876
Log Probability for digit 2 :  -4457.332595925104
Log Probability for digit 3 :  -4455.1015772407745
Log Probability for digit 4 :  -4446.86889622704
Log Probability for digit 5 :  -4479.895689210735
Label:  3
Log Probability for digit 1 :  -4451.760085966908
Log Probability for digit 2 :  -4453.267948208004
Log Probability for digit 3 :  -4439.943401267994
Log Probability for digit 4 :  -4449.413949464036
Log Probability for digit 5 :  -4472.173411277892
Label:  4
Log Probability for digit 1 :  -4438.59965601902
Log Probability for digit 2 :  -4501.691012602742
Log Probability for digit 3 :  -4441.343200368346
Log Probability for digit 4 :  -4450.591000455753
Log Probability for digit 5 :  -4442.626879414615
Label:  5
Log Probability for digit 1 :  -4454.638825667363
Log Probability for digit 2 :  -4464.551580175303
Log Probability for digit 3 :  -4439.135790537592
Log Probability for digit 4 :  -4430.185176688551
Log Probability for digit 5 :  -4467.203539877663
Label:  6
Log Probability for digit 1 :  -4495.411637290953
Log Probability for digit 2 :  -4497.04006442228
Log Probability for digit 3 :  -4446.432408161289
Log Probability for digit 4 :  -4448.540639210706
Log Probability for digit 5 :  -4500.940810787345
Label:  7
Log Probability for digit 1 :  -4426.171758151863
Log Probability for digit 2 :  -4515.582887455524
Log Probability for digit 3 :  -4447.23793079878
Log Probability for digit 4 :  -4443.917138470196
Log Probability for digit 5 :  -4420.041365962184
Label:  8
Log Probability for digit 1 :  -4423.131763142699
Log Probability for digit 2 :  -4472.27485501851
Log Probability for digit 3 :  -4415.322842102786
Log Probability for digit 4 :  -4432.219689629081
Log Probability for digit 5 :  -4442.66421322814
Label:  9
Log Probability for digit 1 :  -4416.616440773744
Log Probability for digit 2 :  -4496.348408913746
Log Probability for digit 3 :  -4438.632386304114
Log Probability for digit 4 :  -4446.975023815595
Log Probability for digit 5 :  -4422.962963344602
```

DSE 210: Probability and statistics Winter 2020

Worksheet 5 — Classification with Generative models 1

*18. **Programming Question**: (15 points total, 3 points each)*

(1) Create a classification problem with 3 classes, 15 features and 5000 rows.

Slide Type

```
# 1. Create a classification problem with 3 classes, 15 features and 5000 rows

from sklearn.datasets import make_classification

X, y = make_classification(n_samples=5000, n_features=15, n_classes = 3, n_informative = 3)
```

Slide Type

```
# print X

pd.DataFrame(X).head(5)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
0	-0.479888	-0.600105	0.303918	1.535755	0.340140	-1.988212	-0.027653	-0.894987	-1.587801	2.467105	-1.136715	-2.083581	-0.380400	1.974672	-0.69
1	0.633348	0.962290	0.814473	-0.003243	0.929666	-0.744475	-1.337174	-0.366015	1.386391	-0.227363	0.760677	-0.936752	-0.685653	-0.756152	-1.65
2	0.933183	-0.612267	-1.034197	0.369370	-1.411523	-0.324664	-0.064488	-0.172099	-2.458331	-1.098262	1.258992	0.611090	-0.705192	-1.617141	-0.26
3	-3.441939	-0.951017	-1.048750	1.443221	-1.250332	0.164343	1.893598	-0.280170	-1.193462	-0.933628	-3.030818	0.813125	4.508485	3.222326	0.47
4	0.385174	-0.315077	-1.458283	-0.506321	0.473403	-0.748647	-2.977722	0.326998	1.128224	0.548446	-0.833508	-1.124286	-0.551719	-0.902384	-1.05

Slide Type

```
# print y

y
```

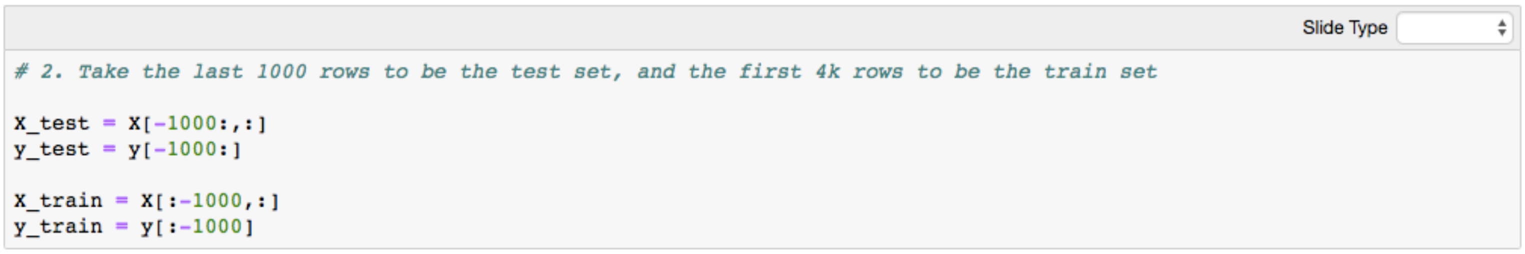
```
array([2, 2, 2, ..., 1, 2, 2])
```

Slide Type

```
# print size of X and y

print(X.shape)
print(y.shape)
```

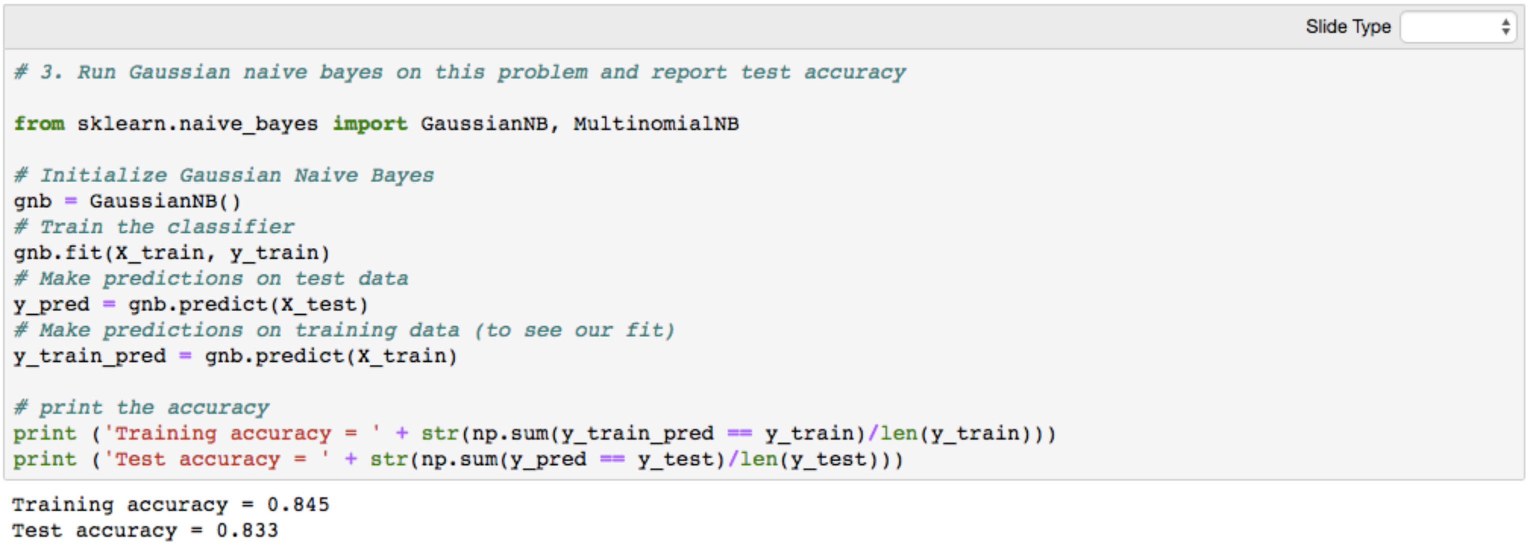
```
(5000, 15)
(5000,)
```

(2) Take the last 1000 rows to be the test set.



```
# 2. Take the last 1000 rows to be the test set, and the first 4k rows to be the train set

X_test = X[-1000:,:]
y_test = y[-1000:]

X_train = X[:-1000,:]
y_train = y[:-1000]
```

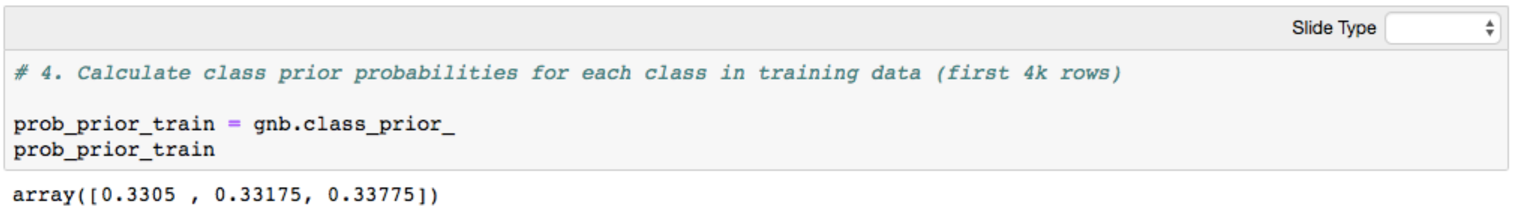
(3) Run Gaussian naive bayes on this problem and report test accuracy.



```
# 3. Run Gaussian naive bayes on this problem and report test accuracy

from sklearn.naive_bayes import GaussianNB, MultinomialNB

# Initialize Gaussian Naive Bayes
gnb = GaussianNB()
# Train the classifier
gnb.fit(X_train, y_train)
# Make predictions on test data
y_pred = gnb.predict(X_test)
# Make predictions on training data (to see our fit)
y_train_pred = gnb.predict(X_train)

# print the accuracy
print ('Training accuracy = ' + str(np.sum(y_train_pred == y_train)/len(y_train)))
print ('Test accuracy = ' + str(np.sum(y_pred == y_test)/len(y_test)))
```

```
Training accuracy = 0.845
Test accuracy = 0.833
```

(4) Calculate class prior probabilities for each class in training data (first 4k rows).



```
# 4. Calculate class prior probabilities for each class in training data (first 4k rows)

prob_prior_train = gnb.class_prior_
prob_prior_train
```

```
array([0.3305 , 0.33175, 0.33775])
```

(5) Calculate the probability of the samples for each class in the test set.

Slide Type

```
# 5. Calculate the probability of the samples for each class in the test set

prob_test = gnb.predict_proba(X_test)
prob_test
```

```
array([[8.72482662e-01, 9.57308927e-02, 3.17864455e-02],
       [2.55717046e-01, 7.14314246e-01, 2.99687078e-02],
       [5.09309066e-08, 9.99996257e-01, 3.69179916e-06],
       ...,
       [9.67700599e-03, 9.89714274e-01, 6.08719607e-04],
       [3.01023719e-02, 8.31934215e-03, 9.61578286e-01],
       [1.96811484e-02, 5.01668944e-02, 9.30151957e-01]])
```

Slide Type

```
# find size of probability for test set
prob_test.shape
```

```
(1000, 3)
```