COSC343/AIML402 S2 2023

Lab 6: Naive Bayes classifier and machine learning

Tutorial objectives:

- **To review the concepts from Bayesian reasoning**
- **To complete a quick tutorial on Numpy and Matplotlib, which are very useful Python libraries for machine learning (learning more about those libraries outside of this tutorial is strongly advised)**
- **To implement a Naive Bayes Classifier**
- **To get a sense of the conventions of the Sklearn library, which is a Python library that implements a lot of machine learning algorithms**
- **To apply the Naive Bayes Classifier to the problem of spam filtering**
- **To get into the habit of training and testing machine learning models on separated train and test sets**

Preparation:

- **Complete "A quick introduction to Numpy and Matplotlib", starting on the next page, taking time to understand what the example code is doing.**

Probabilistic reasoning concepts

1. What does it mean to say that two random variables in a probability model are **independent**?

2. What is **conditional independence**, and why is it useful?

3. Conceptually, what do the terms **prior** and **posterior** correspond to in the Bayes rule?

A quick introduction to Numpy and Matplotlib

Pretty much all machine learning libraries in Python are built on top of the Numpy library, which provides means of working with vectors, matrices and tensors. Matplotlib library is used for plotting things. Hence, we need to do a bit of introduction to Numpy and Matplotlib (which should come in handy for the tutorials that follow and the second assignment).

1. Create a new project in PyCharm (make sure to select the cosc343 environment for the python interpreter). Create a new script, called `exercise1.py`.

2. To import the Numpy library put `import numpy as np` at the top of your Python script. From now on, every call to a method preceded by `np.` will be a call to the Numpy library.

```
import numpy as np
```

3. Let's suppose we are working with the following data:

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

,

where x_1 and x_2 represent two percepts/input attributes, and y corresponding desired action/ label; and we have four different examples of the percepts to label mappings. In Numpy we would represent input as a two-dimensional Numpy array of shape 4×2. The target output y would be specified as a separate Numpy array, a vector of length 4. Here's the syntax for how to create the Numpy arrays corresponding to the tables above:

```
x = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])

y = np.array([0,
              1,
              1,
              0])
```

Rather than thinking of them as matrices and vectors, it's best to think of Numpy arrays (as the name suggests) as arrays of certain shape over some number of dimensions.

4. To examine the shape of the array you use the Numpy's shape function. For example, to print the shape of the two variable you do:

```
print(np.shape(x))

print(np.shape(y))
```

   The output for these print statements should be (4,2) and (4,) respectively. The `np.shape()` function returns a tuple – the length of the tuple corresponds to the dimensionality of the Numpy array and the values in the tuple give the size of each dimension. Thus, `x` is a 2-dimensional array of size 4×2, and `y` is a 1-dimensional array of size 4.

5. When it comes to 2-dimensional arrays, you can think of them as tables – the first dimension is the number of rows and the second dimension is the number of columns. By convention, we organise the data so that the rows correspond to different samples and columns correspond to attributes of each sample. A vector is just a single column table. In the example above, variable `x` contains 4 samples of 2 attributes each, and variable `y` contains 4 samples of 1 attribute. Function `len()` applied to a Numpy array returns the size of the first dimension. So, if you want to know the number of points in a given dataset you can simply do

```
print(len(x))    # equivalent to np.shape(x)[0]

print(len(y))    # equivalent to np.shape(y)[0]
```

6. To select the the nth point from `x`, you do `x[n]` – this syntax is equivalent to "select nth row"...which for x in our example will be a 2-dimensional vector. In the example below, we print the contents of the first row of x and the corresponding label in y.

```
print(x[1])

print(y[1])
```

7. The : specifies all elements from a given dimension. So, to print the first column of `x` you can do:

```
print(x[:,0])
```

8. You can also use negative indexing to index backwards from the end of the array. To print the last point in `x`, and its corresponding label from `y`, you can do:

```
print(x[-1,:])

print(y[-1])
```

Note that `x[-1,:]` is equivalent to `x[-1]`.

9. Awesome thing about Numpy is that arithmetic operations apply to the entire array all at once. Let's say we wanted to normalise `x` to range between -1 and 1. In essence, we want to modify all 0's to -1 and leave all 1's as one. We can accomplish this by multiplying every entry in the array by 2 and then subtracting 1. In Numpy there is no need to loop over all elements of the array, you can just do:

```
x = 2*x-1 # Multiply all entries in x by 2 and then subtract 1

print(x)
```

In the example above, one operand was a scalar and the other a Numpy array.

10. We can also perform operations between two Numpy arrays – if their sizes match, then these operations are done element-wise. For instance, to get a row-wise sum of all the elements in `x` we can do

```
x_row_sum = x[0,:]+x[1,:]+x[2,:]+x[3,:]
```

The resulting `x_row_sum` will be a 1-dimensional array of 2 things (since the size of the second dimension of x is 2). You can also achieve the same results using Numpy's sum function like so:

```
x_row_sum = np.sum(x,axis=0)
```

The above syntax uses the Numpy's sum function, which takes a Numpy array as the first argument and the `axis` argument specifies the index of the dimension (in this case the first dimension) along which to sum.

11. Here are som functions for creating Numpy arrays:

```
x = np.zeros( shape=(50,3) ) #Creates a 2-dim numpy array of zeros,
                             #essentially a 50x3 matrix of zeros


x = np.ones( shape=(45,2) ) #Creates a 2-dim numpy array of ones,
                            #essentially a 45x2 matrix of ones

x = np.random.rand(12,4)    #Creats a 2-dim numpy array of random
                            # values, essentially a 12x4 matrix of
```

```
                                        # random values

x = np.arange(start=2,stop=8,step=0.5)
                                        #Creates a 1-dim numpy array of values
                                        #starting from 2, incrementing by 0.5
                                        # and going up to (but not including) 8.

x = np.linspace(start=0,stop=50,num=13)
                                # Creates a 1-dim numpy array of 13 values
                                # equally spaced between 0 and 50 (but not
                                # including 50)
```

12. Let's turn our attention to plotting. At the top of the file (below the numpy import) import the matplotlib library like so:

```python
import matplotlib.pyplot as plt
```

This syntax imports the pyplot submodule of Matplotlib under the alias `plt`. Let's say we wanted to re-create the scatter plot from Lecture 10.

```python
x = np.array([2001, 2002, 2003, 2004, 2005])
y = np.array([1,4,9,0.5,25])

plt.scatter(x,y,marker='x',color='black')
plt.show()
```

The scatter function of Matplotlib expects the first and the second arguments to be 1-dim Numpy arrays of the same size, and it plots the values of one array against the other. The `marker` and `color` arguments specify the visual marker and its colour for denoting a points in the plot. Matplotlib's show function displays allows plotting many things on the same plot; the function is blocking function, which means that it once it displays the plot, it will "block" the execution of the program and wait for the user to close the plot before resuming execution of the rest of the script. PyCharm disables this behaviour by default, making the call non-blocking and the plot showing up on the side of the PyCharm window. If you want to disable this 'feature' you need to go to `PyCharm->Preferences->Tools->Python Scientific` and disable the "Show plots in tool window" option (on Windows it's "Settings" instead of "Preferences").

13. If you wanted to connect all the points in the plot, you can use Matplotlib's plot method.

```python
plt.plot(x,y,color='blue')
```

If you add this new statement between the previous `plt.scatter` and `plt.show` calls, you'll get a blue line over the points. Here's more (hopefully self-explanatory) syntax for labelling of the various parts of the previous plot.

```
plt.scatter(x,y,marker='x',color='black', label="Scatter")
plt.plot(x,y,color='blue',label="Line")
plt.xlabel('Years')
plt.ylabel('Sales (in K$)')
plt.legend()
plt.show()
```

14. Time to test yourself. Here's a set of simple challenges to test your understanding of Numpy and Matplotlib.

    Let's go back to

```
x = np.array([[0, 0],
              [0, 1],
              [1, 0],
              [1, 1]])

y = np.array([0,
              1,
              1,
              0])

x = 2*x-1
```

    Consider the following statement:

```
z = x[:,0]*x[:,1]
```

    How many dimensions does `z` have and what is the corresponding size of each dimension? What is this computation achieving?

15. Consider the following statement

```
z=np.sum(x,axis=1)
z=z**2
```

    which (by the way) you can also write as

```
z=np.sum(x,axis=1)**2
```

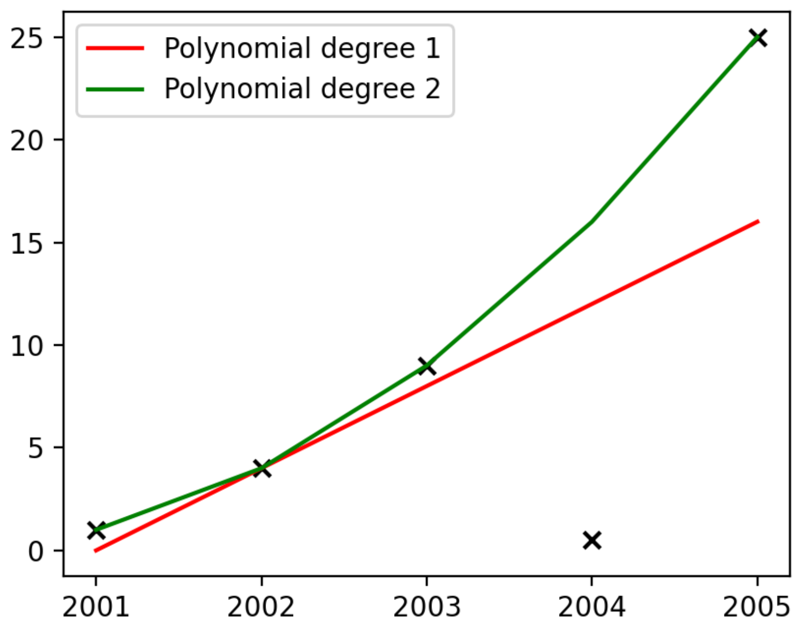
    What's `z` and its contents?

16. Next, consider the data from Lecture 10:

```
x = np.array([2001, 2002, 2003, 2004, 2005])
y = np.array([1,4,9,0.5,25])

plt.scatter(x,y,marker='x',color='black')
plt.show()
```

Compute and plot (as a continuous line) a 1-degree polynomial fit of the data given by the equation $y = w_1 x + w_0$, where $w_1 = 4$ and $w_0 = -8004$. Compute and plot on the same plot (as a continuous line) a 2-degree polynomial fit of the data given by the equation $y = w_2 x^2 + w_1 x + w_0$, where $w_2 = 1$, $w_1 = -4000$ and $w_0 = 4000000$. The resulting plot should look like this:



This is the end of this lightning fast tutorial of the Numpy and Matplotlib libraries. More functions and capabilities will be introduces later as needed. However, it might be a good idea to go over some online tutorials (such as Numpy and/or Matplotlib) to acquaint yourself with further capabilities of those two libraries.

Naive Bayes Classifier

Exercise 1: In this exercise you will be guided through an implementation of a generic, multi-class, bag of words Naive Bayes Classifier. All the code is provided below, but in small chunks, with you are meant to type out (rather than copy and paste). The objective is not just to have a working code, but to go through it slowly and understand how it all works.

The following implementation of the Naive Bayes Classifier emulates the conventions of the Sklearn library. Though the library is not introduced explicitly in this tutorial, it will make frequent appearances in later ones;this exercise should be a great way to understand how Skleanr's API is structured. Assuming you have already created a Python project (in step 1) download the `NaiveBayesClassifier.py` script from Blackboard and place it in your PyCharm project folder. You'll find the following boiler template for our implementation of the NaiveBayesClassifier class.

```
class NaiveBayesClassifier():

  def __init__(self, verbose=False):
    self.verbose = verbose
    self.trained = False

  def fit(self,X,y):

  def predict(self,X):
    if not self.trained:
       raise RuntimeError("Model needs training first.")
```

Sklearn library implements a given machine learning model as a class. The initialiser takes various arguments that set up the aspects of the model that need to be configured before training (in the machine learning lingo these are often referred to as *hyper-parameters*). The `fit()` method of the model implements the training algorithm and the `predict()` method produces the output of the model (once it's trained). The Naive Bayes Classifier we are implementing takes only a single explicit argument (the `self` argument is implicit, in that it doesn't get specified when calling the method of an object), that explicit arguments will control the verbosity of the trining process (and it is set to False by default). The initialiser also creates an internal variable `trained` and sets it to False, to prevent the execution of the `predict` method on an untrained model.

Naive Bayes Classifier is essentially a model that makes decisions on new (test) data based on the (train) data from previous observations, and so it can be thought of as a supervised learning model. Recall (from Lecture 9) that, in order to make a decision, Naive Bayes Classifier compares the likelihoods of the (assumed to be) conditionally independent evidence multiplied by the prior belief (expressed as probability) of the cause. In a supervised classification setting, the class label is equivalent to the "cause" and the data from which to infer the label is the "evidence". For this exercise we will develop a bag of words Naive

Bayes Classifier, where the evidence consists of the composition of values of certain type. An example of a problem that this classifier can solve is a spam filter, which needs to label e-mail messages as either "spam" or "ham" (ham stands for no spam) from the words of the e-mail subject. The words in the subject constitute the evidence. The naive assumption is that the words are conditionally independent given the label. That's in fact not true, because the meaning of a word can be altered by the words around it...but for the purpose of this spam filter, we'll assume they are independent.

In general, we assume a data sample has n pieces of evidence – x_1 through x_n and a single label y:

x_1	x_2	$\dots$	x_n	y

The label indicates the sample as a member of K possible classes $y \in \{c_1, c_2, \dots, c_K\}$. For instance, in the spam filter, the words in the e-mail subject are taken to be individual pieces of evidence and the are two classes – the e-mail is either a "ham" or "spam". Hence, to decided on the label of a given sample, we need to compute

$$P(x_1|c_k)P(x_2|c_k) \cdot \ldots \cdot P(x_N|c_k)P(c_k) \tag{1}$$

for K classes c_1 through to c_K; whichever of the computations gives the largest likelihood, the corresponding label is the predicted label of the sample. The order of x's in does not affect the result of Equation 1 – this is a feature of the "bag of words" model. Of course, we know that order of words in sentences is important for the meaning, but from the Naive Bayes Classifier's point of view the order is arbitrary – it will make its decision based on what words are found in the subject, and not how they are arranged. This is acceptable, as long as the classifier can be shown to be still fairly accurate (and we'll come to the evaluation in a bit).

The "training" of our Naive Bayes classifier, and thus the guts of the `fit` method of the `NaiveBayesClassifier` class, is about computation of the distributions $p(x_i|c_k)$'s and $p(c_k)$'s from the training data, for each c_k in the set $\{c_1, \dots, c_K\}$, so that decisions can be made later about about the classification of new emails as "spam" or "ham" using Equation 1.

By Sklearn's convention, the `fit` method of a supervised learning model takes two arguments: X, which is assumed to be the input data of N training samples, and y, an array of N corresponding labels.

The following bits of code all go into the `fit(self,X,y)` method. We are assuming that `X` is a Numpy array of N lists, each list containing an arbitrary number of data values (for example, words in the e-mail subject) and `y` is a Numpy array of N labels.

The first thing that we need to do is to determine how many classes there are in the data. Sklearn API doesn't require explicit specification of the number of classes, because they can be inferred from the values of y passed into the `fit()` method. For this we will use the Numpy's unique function, like so:

```
self.class_labels, class_counts = np.unique(y,return_counts=True)
```

The first argument of the `unique` function is the array to be examined for unique values, other arguments specify whether other information is to be extracted – in this case, we also want the function to return the frequency of each of the values in `y`. With `return_counts` (and nothing else) set to True, the function returns a tuple, where the first item is a Numpy array vector containing the unique (and sorted) values found in `y` and the second item is a Numpy array of the same length specifying how many times a given item was found in `y`.

For our purposes, we'll find it easier if the `self.class_labels` is list, so we convert it to a list using Numpy's tolist() method

```
self.class_labels = self.class_labels.tolist()
```

Since `y` is meant to provide labels of the data, the `self.class_labels` is now a list of unique labels and its length gives us the number of unique labels/classes.

```
n_classes = len(self.class_labels)
```

Note also that `self.class_labels` is saved into the object instance (that's what the `self.` prefix does), because the model needs to remember the labels in order to produce predictions using those labels.

The computation of the prior distribution $p(c_k)$ for $k = 1, \ldots, K$ is as simple as this:

```
self.priors = class_counts/np.sum(class_counts)
```

Recall that `class_counts` is a Numpy array of length K, containing the frequency counts of labels in `y`. The Numpy sum function adds all the elements of a Numpy array, which in the code above gives the total number of labels in `y`. Dividing a Numpy array by scalar divides every value in that array. And so, in this one statement, we carry out the following computation:

$$\left[\frac{\text{num } c_1\text{'s}}{\text{num all labels}}, \ldots, \frac{\text{nu mc}_K\text{'s}}{\text{num all labels}}\right], \tag{2}$$

which gives an estimation of

$$\left[P(c_1), \ldots, P(c_K)\right], \tag{3}$$

and is saved to `self.priors`.

The next bit is the verbose part that shows the results of the computation of the priors if `self.verbose` is set to True.

```
if self.verbose:
    print("Found %d class labels: %s" %
              (n_classes, str(self.class_labels)))
    print("Priors p(c):")
    for k in range(n_classes):
        class_label = self.class_labels[k]
        prob = self.priors[k]
        print(" P(c='%s')=%.2f" % (class_label,prob))
```

Next, we need to compute the distribution $p(x_i|c_k)$ for each class $k = 1, ...K$. This time around, we'll represent each distribution using a Python dictionary, and we will need K of those distributions.

```
self.prob_data_given_lbl = []
for _ in self.class_labels:
  self.prob_data_given_lbl.append(dict())
```

Now, `self.prob_data_given_lbl` is a list of K dictionaries. Next we need to examine data samples in X and their corresponding labels, counting the number of times a given data value appears in a sample of given class.

```
for sample_index,data_sample in enumerate(X):
  sample_label = y[sample_index]
  k = self.class_labels.index(sample_label)
  for data_val in data_sample:
    if data_val in self.prob_data_given_lbl[k]:
      self.prob_data_given_lbl[k][data_val] +=1
    else:
      self.prob_data_given_lbl[k][data_val] =1
```

The code above iterates over all the samples in X; based on the corresponding label from y, we determine the index of that label in `self.class_labels`, and thus have an index of the dictionary for `self.prob_data_given_lbl` to update. We check if the key corresponding to a given data value found in the sample already exists in the dictionary – if not, that marks the first occurrence of the value in the data of the corresponding label; if the key is found, the frequency count of that data value is incremented by 1. In terms of the spam filtering example, the data sample is the subject line from an e-mail and data values are different words in the subject; we want to count the number of times a given word appears in the subjects of e-mails labelled as "ham", and how many times it appears in the subjects of e-mail labelled as "spam".

After all the counting is done, we need to iterate over all dictionaries and convert the frequency count under each key/word to a probability value. We do this by dividing each value by the total number of emails in training data of the corresponding label.

```
for k in range(len(self.class_labels)):
  for data_val in self.prob_data_given_lbl[k]:
    self.prob_data_given_lbl[k][data_val] /=
                 class_counts[k]
```

And that's that for the training. The `NaiveBayesClassifier` object can be instantiated and trained by invoking its `fit` method. After the `fit` method is done, the object member variables `self.labels`, `self.priors`, and `self.prob_data_given_lbl` contain respectively K class labels, discrete probability distribution $p(c_k)$, and K probability distributions $p(x_i|c_k)$. That's all that is needed for making inferences about the classification of new samples. The last statement of the `fit` method needs to set set the `self.trained` variable of the model to `True`, to allow the use of the `predict` method:

```
self.trained = True
```

Next, let's implement the `predict` method – the following bits of code all go into that method.

The `predict` method takes one explicit argument - X, which we assume is a Numpy array of lists containing data that constitutes the evidence from which to infer the label.

```
num_samples = len(X)
num_classes = len(self.class_labels)
predictions = []
```

The code above also initialises a list of predictions which is to be returned from the `predict` method.

For each sample, we create a likelihood array of length K, where for each $k = 1, ..., K$ we compute the likelihood according to Equation 1.

Since `self.priors` contains prior probabilities $[P(c_1), \ldots, P(c_K)]$ of the computation in Equation 1, we can initialise our likelihood array to the copy of `self.priors`.

```
for sample_index in range(num_samples):
  data_sample = X[sample_index]
  lkhood_class_given_data = np.copy(self.priors)
```

The values of `lkhood_class_given_data` at this point are:

$P(c_1)$
$\vdots$
$P(c_K)$

The following code, which prints some information in verbose mode should be indented and in the scope of the for loop over the samples:

```
  if self.verbose:
    print("Prediction for sample %d" % sample_index)
    print("  Data: %s" % str(data_sample))
    print("  Priors: %s" % str(lkhood_class_given_data))
```

From now on, it's only a matter of going over every value of the data in the data sample (remember, data sample consists of a list of values observed, such as words in an e-mail subject line) and checking in each class dictionary the probability of that value stored under the key equal to the value. If the key of a given value is not found in a given dictionary, the probability of that value for the corresponding class is 0. Note that the code below should be indented in the scope of the for loop over the data samples in `X`:

```
  for data_val in data_sample:
    for k in range(num_classes):
      if data_val in self.prob_data_given_lbl[k]:
        prob = self.prob_data_given_lbl[k][data_val]
      else:
        prob = 0
      lkhood_class_given_data[k] *= prob
    if self.verbose:
      print("  After '%s':" % str(data_val))
      print("  Lkhood: %s" % str(lkhood_class_given_data))
```

Assuming the data sample consists of values $[x_1, \ldots, x_n]$, the values of `lkhood_class_given_data` after the first iteration of `data_val` in `data_sample` will be:

$$\begin{array}{|c|}\hline P(c_1)P(x_1|c_1) \\ \hline \vdots \\ \hline P(c_K)P(x_1|c_K) \\ \hline \end{array},$$

and the values of the `lkhood_class_given_data` after the n^{th} iteration of `data_val` in `data_sample` will be:

$$\begin{array}{|c|}\hline P(c_1)P(x_1|c_1)\dots P(x_n|c_1) \\ \hline \vdots \\ \hline P(c_K)P(x_1|c_K)\dots P(x_n|c_K) \\ \hline \end{array}.$$

At this point, all that needs to be done for classification of the sample is to figure out at which index the value of `lkhood_class_given_data` is the largest. The index corresponds to the label in `self.class_labels` that is to be assigned to the sample. Note that the code below should be indented in the scope of the for loop over the data samples in `X`:

```
k = np.argmax(lkhood_class_given_data)
class_label = self.class_labels[k]
predictions.append(class_label)

if self.verbose:
    print("  Predct: %s" % class_label)
```

And at this point, the big loop over the data samples in `X` is finished. As the final touch, we return the list of predictions.

```
return predictions
```

Spam filter

Exercise 2: In this exercise you will use the `NaiveBayesClassifier` implemented in the previous section to create an e-mail spam filter.

The training data will consists of e-mail subjects that have been labelled as spam or ham:

Label	Subject
ham	question about pca with prcomp
spam	do not miss opportunity to be in perfect form
spam	urgent notification from protection department
ham	accuweather.com alert forecast
⋮	⋮

You need to download a corpus of e-mails from https://plg.uwaterloo.ca/~gvcormac/treccorpus07/. Accept the user agreement and download the 255MB file `trec07p.tgz` into your project's root folder. Once `trec07p.tgz` is downloaded, extract it inside your project folder. Once you extract the contents of `trec07p.tgz` you might want to right click on the trec07p folder in your PyCharm's project and select "Marks Directory as → Excluded" - this will prevent PyCharm from indexing that folder, which would slow PyCharm down since that folder contains lots of files (and they don't need to be indexed).

If you haven't already, download `dataset_trec.py` from Blackboard into your project folder. Create a new Python file and start at the top with the following imports:

```
from NaiveBayesClassifier import NaiveBayesClassifier
from dataset_trec import dataset_trec
import numpy as np
```

The provided `dataset_trec` file processes all the e-mails in the dataset, extracting subjects and converting them to a list of words. To bring the data into your script add the following code:

```
trec07 = dataset_trec.load()

S = trec07.subjects
X = trec07.subject_words
y = trec07.labels
```

Variable `S` contains all the subject lines (about 70K in total), `X` is a list of e-mail subjects, each a list of words; `y` is the list of labels, identifying each subject as either "spam" or "ham". In Lecture 10 we talked about the need to divide the data into a training and test set. Let's split the data into the train set of 1000 e-mail subjects, and leave the rest for testing.

```
num_samples = len(X)
num_train_samples = 1000
```

```
num_test_samples = num_samples-num_train_samples
print("Training on %d samples." % num_train_samples)
print("Testing on %d samples." % num_test_samples)
```

We want the 1000 training samples to be selected at random, and this is how to do this, using Numpy's random.permutation function:

```
random_indices = np.random.permutation(num_samples)
test_sample_indices = random_indices[:num_test_samples]
train_sample_indices = random_indices[num_test_samples:]
```

The first line in the code above creates a random permutation of all indices from `0` to `num_samples-1`. The syntax `[:num_test_samples]` selects the first `num_test_samples` from a Numpy array, whereas the syntax `[num_test_samples:]` selects all the remaining samples starting at index `num_test_samples`. We have now divided an array of unique and randomly shuffled indices into two arrays `test_sample_indices` and `train_sample_indices`. If we convert `X` to a Numpy array of lists and `y` to a Numpy array of strings, we can simply select the items from the arrays using `test_sample_indices` and `train_sample_indices` like so:

```
X = np.array(X, dtype=list)
y = np.array(y, dtype=str)

X_train = X[train_sample_indices]
y_train = y[train_sample_indices]

X_test = X[test_sample_indices]
y_test = y[test_sample_indices]
```

Next, we instantiate `NaiveBayesClassifier` in verbose mode and fit the model using the `X_train` and `y_train` data.

```
model = NaiveBayesClassifier(verbose=True)
model.fit(X_train,y_train)
```

To test the model, we'll use it for prediction of the label of the first sample of X (we are using `X` at this point, not `X_test`, so that you get the same subject line that I did - `X_train` and `X_test` are shuffled, but `X` is not).

```
y_predict = model.predict([X[0]])
```

Why the weird syntax above `[X[0]]` and not just `X[0]`? That's because the prediction method (by Sklearn's convention, which we followed in the previous section) expects a list of data samples...and `X[0]` is just a single sample. So, in order for `predict` method to work properly, we pass a list that contains a single subject line `X[0]`.

At this point you should run your program and examine the output. Note the priors on the "spam" and "ham" labels as well as the results of the prediction.

Now, add the following code, after the prediction on `[X[0]]`, to disable the verbose mode of the classifier:

```
model.verbose = False
```

Next, add the code to get Naive Bayes Classifier predictions on the entire test set and compute their accuracy.

```
y_predict = model.predict(X_test)
accuracy = np.mean(y_predict == y_test)
print("\nNaive Bayes Classifier accuracy on %d test samples: %.2f\n" %
          (num_test_samples,accuracy))
```

Since `y_test` is a Numpy array, and the length of `y_predict` matches the length of `X_test`, the length of `y_test` and `y_predict` are the same. The syntax `y_predict == y_test` compares the items in the Numpy array `y_test` to the items of `y_predcit`, item by item, producing True (1) when corresponding items are the same and False (0) when they are not. Numpy's mean function computes the average value of the resulting Numpy array of 1's and 0's, which gives the accuracy of the classification as a fraction[1].

Run the program. Does your output looks something like this?

```
Training on 1000 samples.
Testing on 70540 samples.
Found 2 class labels: ['ham', 'spam']
Priors:
 p(ham)=0.34
 p(spam)=0.66
Prediction for sample 0
  Data: ['generic', 'cialis', 'branded', 'quality']
  Labels: ['ham', 'spam']
```

[1]If you want to compute classification error, it's as simple as changing `==` to `!=`.

```
  Priors: [0.337 0.663]
  After 'generic':
  Lkhood: [0.    0.003]
  After 'cialis':
  Lkhood: [0.00000000e+00 6.33484163e-05]
  After 'branded':
  Lkhood: [0.00000000e+00 2.86644418e-07]
  After 'quality':
  Lkhood: [0.00000000e+00 3.89110069e-09]
  Predct: spam

Naive Bayes Classifier accuracy on 70540 samples: 0.59
```

Note how poor the accuracy is - only 0.59, meaning the classifier was only correct on 59% of the data. This is bad, because, given the prior $p(\text{spam}) = 0.66$, if the classifier predicted "spam" for every single test sample, it would be correct around 66% of the time. You might think that the problem is the small training sample of 1000 emails. But that's not it. The clue for what the problem is is in the likelihood value for 'ham' going to 0 after the word 'generic'. This word doesn't appear in any of the training 'ham' messages, and so $P(x = \text{"generic"}|c = \text{"ham"}) = 0$ – no matter what other evidence there is, now matter how high the probability of different words appearing in "ham" messages is, the overall likelihood will be zero. It's likely that a single instance of a word not found in training data makes the likelihood of both class labels 0. So, in order to change this, we need to go back to the `predict` method. Find the place where the probability of word is set to 0 if the word is not found in the dictionary `self.prob_data_given_lbl[k]` and instead of making `prob` zero, make it a very small, but larger than zero, number...like 1×10^{-8}:

```python
if data_val in self.prob_data_given_lbl[k]:
  prob = self.prob_data_given_lbl[k][data_val]
else:
  prob = 1e-8
```

Now run your script. Has the accuracy of classification on test data improved? Examine the cumulative 'Lkhood' of in the classification of ['generic','cialis','branded','quality']. Note how small those values get. That's bordering on minimal precision of the float number that computer can represent. For subjects with more words, this in fact might get so small, that likelihood becomes zero. There's a math trick however that can help.

Recall that inference about the label is based on the maximum value of the following:

$$\begin{array}{|c|}\hline P(c_1)P(x_1|c_1)\dots P(x_n|c_1) \\ \hline \vdots \\ \hline P(c_K)P(x_1|c_K)\dots P(x_n|c_K) \\ \hline \end{array}.$$

Well, for the purposes of finding k that gives the maximum in the above table, it doesn't matter what the actual values are, only which one is maximum. So, if we apply a monotonically increasing function, a function that preserves relative order of sizes of all the values, to all the entries, the maximum remains at the same index. The 'log' function is such a function. Hence, finding the maximum of the above table is equivalent to finding the maximum of:

$$\begin{array}{|c|}\hline \log\Big(P(c_1)P(x_1|c_1)\dots P(x_n|c_1)\Big) \\ \hline \vdots \\ \hline \log\Big(P(c_K)P(x_1|c_K)\dots P(x_n|c_K)\Big) \\ \hline \end{array},$$

and by the rules of the 'log' function $\log\Big(\prod x\Big) = \sum \log(x)$, and so we can evaluate the maximum of:

$$\begin{array}{|c|}\hline \log P(c_1) + \log P(x_1|c_1) + \ldots + \log P(x_n|c_1) \\ \hline \vdots \\ \hline \log(P(c_K) + \log P(x_1|c_K) + \ldots \log P(x_n|c_K) \\ \hline \end{array},$$

Again, in the `predict` method, find the place were the `lkhood_class_given_data` is initialised, and this time initialised it to:

```
lkhood_class_given_data = np.log(self.priors)
```

Numpy's log function will do a log on each element of the Numpy array.

Also, where the current value of `lkhood_class_given_data[k]` is multiplied by `prob`, instead of multiplying, add the log value of the `prob`.

```
lkhood_class_given_data[k] += np.log(prob)
```

Run the code again. The test accuracy should be similar, but note the values in 'Lkhood' – they are negative... but that doesn't matter, since they don't change which one is larger (between the "ham" and "spam" side). The important thing is that they are now numbers that are not so small as to get anywhere close to the precision of what the computer can represent. Taking a log of a likelihood is a common practice...and thus the likelihood is often referred to as the *log likelihood*.

```
Training on 1000 samples.
Testing on 70540 samples.
Found 2 class labels: ['ham', 'spam']
Priors:
 p(ham)=0.37
 p(spam)=0.63
Prediction for sample 0
  Data: ['generic', 'cialis', 'branded', 'quality']
  Labels: ['ham', 'spam']
  Priors: [-1.00239343 -0.45728486]
  After 'generic':
  Lkhood: [-19.42307417  -5.80914299]
  After 'cialis':
  Lkhood: [-37.84375492  -9.48702469]
  After 'branded':
  Lkhood: [-56.26443566 -15.24434793]
  After 'quality':
  Lkhood: [-74.68511641 -19.49759378]
  Predct: spam
.
.
.
```

Increase the number of training samples to 10000, see if the test accuracy improves.

```
num_train_samples = 10000
```

As a demonstration of how to use the "trained" Naive Bayes Classifier to label new e-mails, here's the code that processes 20 e-mail subjects (from the test set) into words (using provided method from `trec07` called `subject_to_word_list()`) and makes an inference about the label. Since for this test data we have the actual label, we can compare model's prediction to the *true* answers.

```
S_test = np.array(S_test)
S_test = S[test_sample_indices]
```

```
N = 20

for n in range(N):
    print("Email %d" % (n+1))
    subject = S_test[n]
    print(" %s" % subject)
    wordlist = trec07.subject_to_word_list(subject)
    print(" words=" + str(wordlist))
    y_hat = model.predict([wordlist])
    print(" prediction=" + y_hat[0])
    print(" truth=" + y_test[n])
    print("")
```