Georgia Institute of Technology

ECE 4803: Fundamentamentals of Machine Learning (FunML)

Spring 2022

Homework Assignment # 2

Due: Friday, February 04, 2022 @8PM

Please read the following instructions carefully.

- The entire homework assignment is to be completed on this `ipython` notebook. It is designed to be used with `Google Colab`, but you may use other tools (e.g., Jupyter Lab) as well.
- Make sure that you execute all cells in a way so their output is printed beneath the corresponding cell. Thus, after successfully executing all cells properly, the resulting notebook has all the questions and your answers.
- Print a PDF copy of the notebook with all its outputs printed and submit the **PDF** on `Canvas` under Assignments.
- Make sure you delete any scratch cells before you export this document as a PDF. Do not change the order of the questions and do not remove any part of the questions. Edit at the indicated places only.
- Rename the PDF according to the format: ***LastName_FirstName_ECE_4803_sp22_assignment_#.pdf***
- It is encouraged for you to discuss homework problems amongst each other, but any copying is strictly prohibited and will be subject to Georgia Tech Honor Code.
- Late homework is not accepted unless arranged otherwise and in advance.
- Comment on your codes.
- Refer to the tutorial and the supplementary/reading materials that are posted on `Canvas` for lectures 3, 4, and 5 to help you with this assignment.
- **IMPORTANT:** Start your solution with a **BOLD RED** text that includes the words *solution* and the part of the problem you are working on. For example, start your solution for Part (c) of Problem 2 by having the first line as: **Solution to Problem 2 Part (c)**. Failing to do so may result in a *20% penalty* of the total grade.

Assignment Objectives:

- Learn the fundamentals behind Naïve Bayes and Logistic Regression from both the theoretical and implemntation standpoints
- Learn the use of classes in Python
- Learn the use of performance evaluation metrics for classification tasks

Guide for Exporting Ipython Notebook to PDF:

Here is a video summarizes how to export Ipythin Notebook into PDF.

- **[Method1: Print to PDF]**
  After you run every cell and get their outputs, you can use **[File] -> [Print]** and then choose **[Save as PDF]** to export this Ipython Notebook to PDF for submission.
  Note: Sometimes figures or texts are splited into different pages. Try to tweak the layout by adding empty lines to avoid this effect as much as you can.
- **[Method2: colab-pdf script]**
  The author of that video provided an alternative method that can generate better layout PDF. However, it only works for Ipythin Notebook without embedded images.
  How to use: Put the script below into cells at the end of your Ipythin Notebook. After you run the fisrt cell, it will ask for google drive permission. Executing the second cell will generate the PDF file in your google drive home directory. Make sure you use the correct path and file name.

```
## this will link colab with your google drive
from google.colab import drive
drive.mount('/content/drive')


%%capture
!wget -nc https://raw.githubusercontent.com/brpy/colab-pdf/master/colab_pdf.py
from colab_pdf import colab_pdf
colab_pdf('LastName_FirstName_ECE_4803_sp22_assignment_#.ipynb') ## change path anc
```

Note: Georgia Tech provides a student discount for Adobe Acrobat subscription. Further information can be found here.

Problem 1: Naïve Bayes for Classifying Real-Valued Data (40pts)

In Lecture 4, we learnt the Naïve Bayes classification algorithm to classify discrete-valued feature data. In this problem, we extend this to real-valued data with the popular `Iris` dataset provided by the `sklearn` library. To summarize Naïve Bayes algorithm and implementation, here is **a step-by-step guide**:

0. inspect the dataset and view it
1. write down the Bayes Theorem equation
2. apply/utilize the naïve conditional independence assumption
3. calculate the prior probabilities using the dataset
4. model the likelihood and caluclate model parameters using the dataset. Now you have all components you need for Naïve Bayes classifier.
5. use the likelihood and the priori to calculate posterior probabilities for the test data.
6. categorize the test data according to the highest posterior probability value

In Homework 1, we studied the feature (input) and the target (output) of the `Iris` dataset.

[Execute this cell below]

```
# pip install sklearn  (keep commented if sklearn already installed)
import numpy as np
from sklearn.datasets import load_iris

iris = load_iris()

print('Number of feature is: ' + str(len(iris['feature_names'])) + '\n')

print('Feature names are:')
print(iris['feature_names'])

print('\nNumber of target classes is: ' + str(len(iris['target_names'])) + '\n')

print('Class names are:')
print(iris['target_names'])
```

```
Number of feature is: 4

Feature names are:
['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']

Number of target classes is: 3

Class names are:
['setosa' 'versicolor' 'virginica']
```

Now, we know that there are 4 features and 3 classes in `Iris` dataset, whereas $\mathbf{x} = [x_1, x_2, x_3, x_4]^T \in \mathbb{R}^4$ and $y \in \{c_0, c_1, c_2\}$, $\mathbf{x}$ being a 4-dimensional feature vector for a data point and y being the corresponding label taking on values from the set $\{c_i\}_{i=0}^{2}$.

Problem 1 (a) Review of Bayes Theorem (5pts)

For simplicity, we **only** take the first feature and the first two classes for this part of the problem, part 1(a). **That is, we are trying to use only "sepal length" feature to decide if a flower belongs to "setosa" or "versicolor" classes.** According to Bayes Theorem, we have: (take c_1 as example)

$$P(y = c_1 | x_1) = \frac{P(x_1 | y = c_1) P(y = c_1)}{P(x_1)}$$

The left hand side $P(y = c_1 | x_1)$ basically tells us that **given a certain feature x_1 ("sepal length"), the probability that this flower belongs to class $\mathbf{c_1}$ ("versicolor").** Also, we have $P(y = c_1 | x_1) + P(y = c_0 | x_1) = 1$, since this flower either belongs to "setosa" or "versicolor" class.

Questions: In the above equation, which of the terms does correspond to the likelihood, the marginal likelihood, the prior probabilites, and the posterior probabilities?

What does $P(x_1 | y = c_1)$ mean, in plain english, in this part of the problem?

Solution to Problem 1 Part (a)

- Likelihood: $P(x_1 | y = c_1)$
- Marginal Likelihood: $P(x_1)$
- Prior Probabilities: $P(y = c_1)$
- Posterior Probabilities: $P(y = c_1 | x_1)$
- $P(x_1 | y = c_1)$ means that given a certain class $\mathbf{c_1}$ ("versicolor"), the the probability that this flower has the feature x_1 ("sepal length")

Problem 1 (b) Naïve Bayes and Priori (5pts)

Now, we go back to original 4 features and 3 classes setup in `Iris`, the Bayes theorem becomes:

$$P(y = c_k | \mathbf{x}) = \frac{P(\mathbf{x} | y = c_k) P(y = c_k)}{P(\mathbf{x})}$$

Note that $\mathbf{x}$ is boldface which means it represnets a vector of feature. Since $P(\mathbf{x})$ is a multivariate distribution of 4 features, modeling it is not an easy task. This is where the **naïve assumption** comes into play—it assumes features are **conditionally independent** of each other. Hence we can decouple the likelihood:

$$P(y = c_k|\mathbf{x}) = \frac{\prod_{j=1}^{4} P(x_j|y = c_k)P(y = c_k)}{P(\mathbf{x})}$$

you can also observe that the denominator $P(\mathbf{x})$ is merely a scaling constant and does not depend on y. Therefore, we usually ignore that for our purposes and to simplify the calculations. The above formulation for Naïve Bayes then reduces to:

$$P(y = c_k|\mathbf{x}) \propto \prod_{j=1}^{4} P(x_j|y = c_k) \times P(y = c_k)$$

Now, you can clearly notice that in order to get the posterior probabilities, we need each likelihood and the prior probabilities. We first look at the prior probabilities so that we can simply estimate $P(y = c_k)$ by calculating the ratio of each c_k in the `Iris` dataset. That is $P(y = c_k) = \frac{N_{c_k}}{N}$, where N_{c_k} is the number of examples in c_k class, and N is the number of total examples.

Question: Complete the cell below that calculates the total number of examples N in `Iris`, how many examples in each class N_{c_k}, and the value of $P(y = c_k)$ for each k? [You can use the table below to help yourself build the python implementation in the cell below]

Solution to Problem 1 Part (b)

$P(y=0)$	$P(y=1)$	$P(y=2)$

```
labels = iris.target

'''
N: an integer value
  Number of total examples in Iris dataset.

N_ck: ndarray of shape (4,)
  Number of examples in each class.
```

```
    P_yck: ndarray of shape (4,)
      Priori of each class.
    '''

    #-----------------Don't change anything above------------------------#

    N = iris.target.shape[0]

    N_ck = np.bincount(iris.target)

    P_yck = np.divide(N_ck, N)

    #-----------------Don't change anything below------------------------#
    print('Total number of examples in Iris: N =', end =' ')
    print(N)

    for i in range(3):
      print('Number of examples in class ' + str(i) + ': N_c' +str(i)+ ' =', end =' ')
      print(N_ck[i])

    for i in range(3):
      print('P(y=c' + str(i) + ') =', end =' ')
      print(P_yck[i])
```

```
Total number of examples in Iris: N = 150
Number of examples in class 0: N_c0 = 50
Number of examples in class 1: N_c1 = 50
Number of examples in class 2: N_c2 = 50
P(y=c0) = 0.3333333333333333
P(y=c1) = 0.3333333333333333
P(y=c2) = 0.3333333333333333
```

Problem 1 (c) Probabilistic Model for Likelihood (10pts)

Next, we want to link our likelihood to probabilistic models. Each of the $P(x_j|y=c_k)$ terms conditions a feature variable on a target class via a probability distribution parameterized by certain parameters, θ. These parameters could be the mean and standard deviation of a Gaussian distribtuion in case of real-valued features, or they could just be the success rate in a Bernoulli distribution in case of binary valued x_j. In any case, these parameters are learnt during the training process from the labeled training data. It would then be more appropriate to write the above formulation as:

$$P(y=c_k|\mathbf{x}) \propto \prod_{j=1}^{4} P(x_j|y=c_k, \theta_{jc_k}) \times P(y=c_k),$$

where θ_{jc_k} represents the parameter set characterizing the conditional distribution of feature j on class c_k.

Question: We load a few examples in `Iris` and assume their features to be independent of each other and sampled from **Gaussian distributions**. Say $\hat{\mu}_{jc_k} = \frac{\sum_{i=1}^{N} x_{ij} \times 1_{y_i=c_k}}{N_{c_k}}$ and $\hat{\sigma}_{jc_k} = \sqrt{\frac{\sum_{i=1}^{N} (x_{ij} - \hat{\mu}_{jc_k})^2 \times 1_{y_i=c_k}}{N_{c_k}}}$. $\mu_{jc_k}, \sigma_{jc_k}$ refer to the mean and st. deviation, respectively, of the $j-th$ feature variable in class c_k, i refers to the training example number, and 1_{cond} is the identity function that takes the value 1 when the $cond$ is true and 0 otherwise. Complete the cell below that calculates values of the mean, $\hat{\mu}$, and std, $\hat{\sigma}$, for every feature and for each class. [You can use the table below to help yourself build the python implementation in the cell below]

Solution to Problem 1 Part (c)

	$\hat{\mu}_{1y}$	$\hat{\mu}_{2y}$	$\hat{\mu}_{3y}$	$\hat{\mu}_{4y}$
$y = 0$				
$y = 1$				
$y = 2$				

	$\hat{\sigma}_{1y}$	$\hat{\sigma}_{2y}$	$\hat{\sigma}_{3y}$	$\hat{\sigma}_{4y}$
$y = 0$				
$y = 1$				
$y = 2$				

```
features = iris.data[1::17, :]
labels = iris.target[1::17]

print('A subset of the Iris dataset:')
print('example number  x1  x2  x3  x4   y')
for i in range(9):
  print('      '+ str(i+1), end ='        ')
  print(features[i, :], end =' ')
  print(labels[i])

'''
mu: ndarray of shape (3, 4)
  Numpy array containing mu_jck.

sigma: ndarray of shape (3, 4)
  Numpy array containing sigma_jck.
'''
mu = np.zeros((3, 4))
sigma = np.zeros((3, 4))
```

```
#-----------------Don't change anything above------------------------#

mu = np.concatenate(([np.mean(features[labels == 0,:], axis=0)], [np.mean(features[labels ==

sigma = np.concatenate(([np.std(features[labels == 0,:], axis=0)], [np.std(features[labels ==


#-----------------Don't change anything below------------------------#
with np.printoptions(precision=8, floatmode='fixed'):
  print('\nAnswer:')
  print('     mu_1y      mu_2y      mu_3y      mu_4y')
  for i in range(3):
    print('y=' + str(i), end =' ')
    print(mu[i])

  print('\n     sigma_1y   sigma_2y   sigma_3y   sigma_4y')
  for i in range(3):
    print('y=' + str(i), end =' ')
    print(sigma[i])
```

```
A subset of the Iris dataset:
example number  x1  x2  x3  x4   y
      1        [4.9 3.  1.4 0.2] 0
      2        [5.7 3.8 1.7 0.3] 0
      3        [5.  3.2 1.2 0.2] 0
      4        [6.9 3.1 4.9 1.5] 1
      5        [5.6 2.5 3.9 1.1] 1
      6        [6.7 3.1 4.7 1.5] 1
      7        [6.3 2.9 5.6 1.8] 2
      8        [6.9 3.2 5.7 2.3] 2
      9        [6.4 3.1 5.5 1.8] 2

Answer:
     mu_1y      mu_2y      mu_3y      mu_4y
y=0 [5.20000000 3.33333333 1.43333333 0.23333333]
y=1 [6.40000000 2.90000000 4.50000000 1.36666667]
y=2 [6.53333333 3.06666667 5.60000000 1.96666667]

     sigma_1y   sigma_2y   sigma_3y   sigma_4y
y=0 [0.35590261 0.33993463 0.20548047 0.04714045]
y=1 [0.57154761 0.28284271 0.43204938 0.18856181]
y=2 [0.26246693 0.12472191 0.08164966 0.23570226]
```

Problem 1 (d) Inference (10pts)

Finally, we have our classifier trained on the subset of `Iris`, and we can use this model for inferencing. The inference phase, also referred to as testing phase, is carried out by obtaining the likelihoods of all test data points by sampling them from the parameterized distributions learnt earlier, and then maximizing the posterior over all possible labels, as shown below:

$$\hat{y} = \underset{c_k}{\text{argmax}} \prod_{j=1}^{4} P(x_j^{test} | y = c_k, \theta_{jc_k}) \times P(y = c_k)$$

The $\hat{y} = c_k$ associated with the highest posterior probility is the classified result of the test data. This is called the **Maximum a posteriori (MAP) estimation**. Alternatively, the prior may be set to be uniform, in which case the formulation reduces to just maximizing the conditional data likelihoods, in what is known as the **Maximum Likelihood Estimation (MLE)**.

An issue that frequently occurs with long chains of probability products is that of numerical underflow i.e., the computer is unable to handle extremely high levels of precision required and forces the result to just be zero. This is often circumvented by computing the logs of the probabilities rather than the raw probabilities themselves, turning the product chain into a summation chain. The maximization may then be carried out in the log space (e as base). This is made possible by the monotonic behavior of the log function—what minimizes or maximizes $f(x)$ also minimizes or maximizes, respectively, $\log f(x)$. The restructured formulation is given below:

$$\hat{y} = \underset{c_k}{\text{argmax}} \sum_{j=1}^{4} \log P(x_j^{test} | y = c_k, \theta_{jc_k}) + \log P(y = c_k)$$

To obtain the posteriors back, one may always exponentiate the expression on the right hand side, and normalize afterwards.

Questions: In problem 1 (b), we calculated $P(y = c_k)$. In problem 1 (c), we calculated μ_{jc_k} and σ_{jc_k} for the Gaussian likelihood model, $P(x_j | y = c_k) = \mathcal{N}(\mu_{jc_k}, \sigma_{jc_k}^2)$. You will need to use those numbers to answer the following questions.

i) Write down the log form of the likelihood function. In other words, what is $\log P(x_j | y = c_k)$? Then complete the `log_gaussian` function.

ii) Suppose you are given a test data sample, flower with feature $\mathbf{x}^{test} = [5, 3, 1, 0.5]^T$. Compute the posteriors for this given sample by hand, calculator, or numpy. Then, complete the calculation of the variable `posteriors` in the cell below.

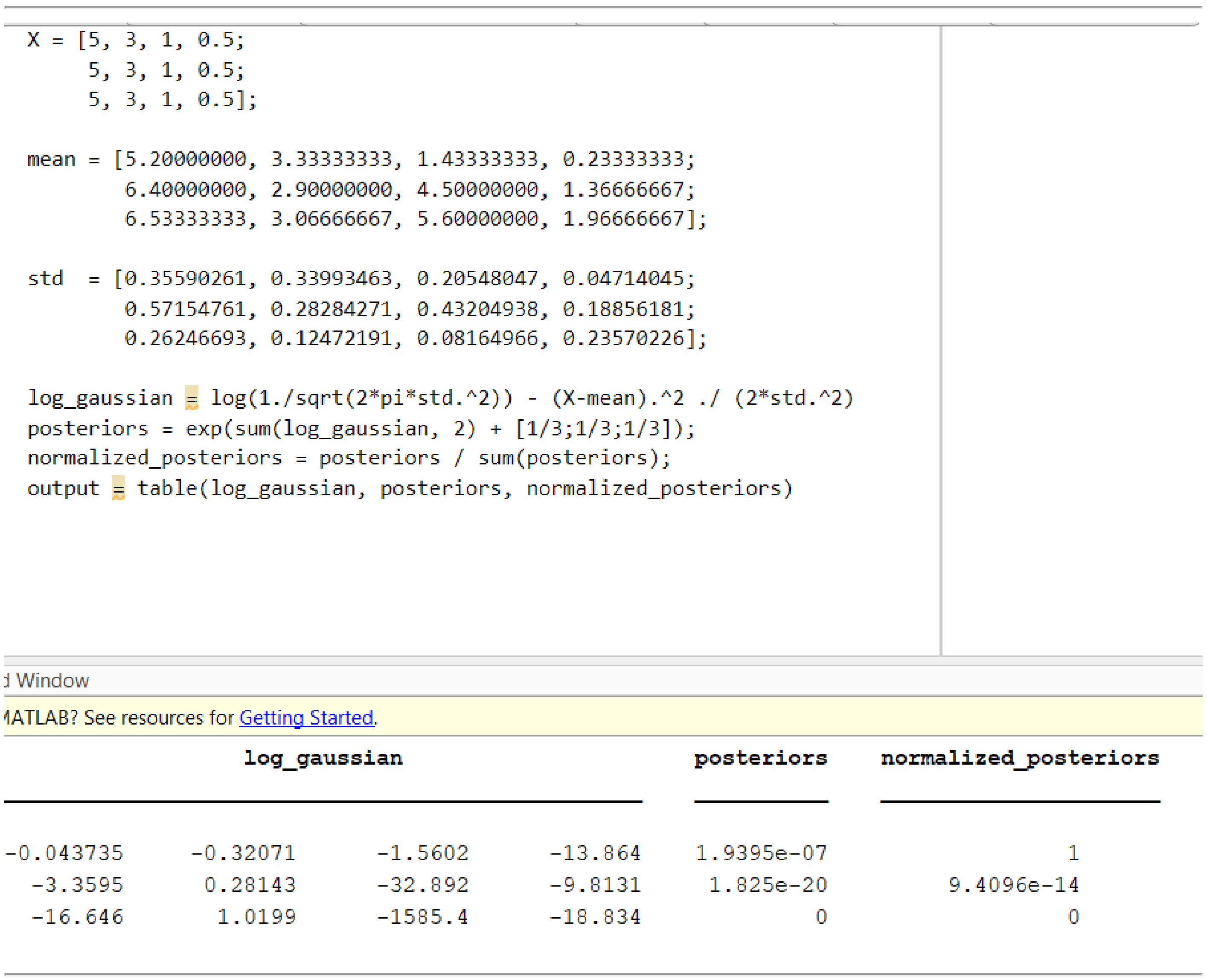
iii) Finally, according to the result above, to which class does this test example belong? Also, complete the calculation of the variable `y_hat` in the cell below.

[You should answer this question both in a text cell and the code cell below]

Solution to Problem 1 Part (d)

i) $\log(P(x_j|y=c_k)) = \log(\frac{1}{\sqrt{2\pi\sigma_{y_k}^2}}\exp{(-\frac{(x_j-\mu_{y_k})^2}{2\sigma_{y_k}^2})}) = \log(\frac{1}{\sqrt{2\pi\sigma_{y_k}^2}}) - \frac{(x_j-\mu_{y_k})^2}{2\sigma_{y_k}^2}$

ii) Hand Calculation of posteriors using MATLAB as calculator:



```
X = [5, 3, 1, 0.5;
     5, 3, 1, 0.5;
     5, 3, 1, 0.5];

mean = [5.20000000, 3.33333333, 1.43333333, 0.23333333;
        6.40000000, 2.90000000, 4.50000000, 1.36666667;
        6.53333333, 3.06666667, 5.60000000, 1.96666667];

std  = [0.35590261, 0.33993463, 0.20548047, 0.04714045;
        0.57154761, 0.28284271, 0.43204938, 0.18856181;
        0.26246693, 0.12472191, 0.08164966, 0.23570226];

log_gaussian = log(1./sqrt(2*pi*std.^2)) - (X-mean).^2 ./ (2*std.^2)
posteriors = exp(sum(log_gaussian, 2) + [1/3;1/3;1/3]);
normalized_posteriors = posteriors / sum(posteriors);
output = table(log_gaussian, posteriors, normalized_posteriors)
```

d Window

ATLAB? See resources for Getting Started.

```
                  log_gaussian                       posteriors    normalized_posteriors
__________________________________________________   __________    _____________________

-0.043735    -0.32071     -1.5602     -13.864        1.9395e-07                1
  -3.3595     0.28143     -32.892     -9.8131         1.825e-20       9.4096e-14
  -16.646      1.0199     -1585.4     -18.834                 0                0
```

iii) Based on the value of y hat, they belong to c_0.

```
x_test = np.array([[5, 3, 1, 0.5]])

'''
posteriors: ndarray of shape (3,)
  Numpy array containing posteriers for each class

y_hat: an integer value
  Classified result of the test data.

'''


#----------------Don't change anything above-----------------------#
```

```
## question i
def log_gaussian(x, mean, std):
  """Function computes log P(x) from a normal distribution specified by
  parameters mean and std."""

  log_likelihood = np.log( 1/np.sqrt(2*(np.pi)*np.square(std)) ) - (np.square(x-mean))/(2*np.

  return log_likelihood


## question ii
posteriors = np.exp( np.sum(log_gaussian(x_test, mu, sigma), axis = 1) + np.log(P_yck) )
posteriors = posteriors / np.sum(posteriors)

## question iii
y_hat = np.argmax(posteriors, axis = 0)

#-----------------Don't change anything below------------------------#
print('y_hat=', end=' ')
print(y_hat)

print('posteriors =', end=' ')
print(posteriors)
```

```
     y_hat= 0
     posteriors = [1.00000000e+00 9.40957643e-14 0.00000000e+00]
```

Problem 1 (e) Naïve Bayes Classifier Implementation (10pts)

Finally, you are going to implement your very own Naïve Bayes classifier, with its `fit()` and `predict()` functions, among others. We will work with the `Iris` dataset as an example, but the class should be able to take as input any other real valued feature data of any number of features and training examples, and be able to predict classes based on the MAP principle for unseen test data, as well as return the normalized posterior probabilities. Since we are working with real-valued data, we are going to impose the conditional feature distributions to be gaussians parameterized by two paramaters, mean (μ) and standard deviation (σ).

i) Complete the `log_gaussian()` function. This should be similar to problem 1 (d).

ii) Complete the `fit()` function. This function fits the model parameters to the dataset. This should be similar to problem 1 (b) and (c).

iii) Complete the `predict` function. It uses the parameters calculated in ii), perform inference on test data by computing the posterior probabilities for each point in the test set and then selecting the class corresponding to the highest posterior. This should be similar to problem 1 (d).

> Do not change the function definitions for the functions defined in the *MyNaiveBayes* class template below. They should take inputs and output results of the form indicated. You are free to add other internal functions and use them inside the class definition as you see convenient. However, that should not change the external code's structure, nor the shape and form of the outputs returned. ***Note***: Any variable preceded by the `self.` keyword gets stored by the class structure and can be used and changed afterwards inside the class regardless of whether the function that first made it returns it or not.

Solution to Problem 1 Part (e)

```
# implement naive bayes class for iris

import numpy as np
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

class MyNaiveBayes:
    def __init__(self, X_train, y_train):
        """Function intializes the Naive Bayes class.

        Parameters:
        -----------
        X_train: ndarray of shape (N,D)
            Numpy array containing N training examples, each D dimensional.

        y_train: ndarray of shape (N,)
            Numpy array containing vector of ground truth classes for examples
            in X_train
        """

        self.X_train = X_train
        self.y_train = y_train

    ## quetion i
    def log_gaussian(self, x, mean, std):
        """Function computes log P(x) from a normal distribution specified by
        parameters mean and std. To be called during inference"""


        log_likelihood = np.log( 1/np.sqrt(2*(np.pi)*np.square(std)) ) - (np.square(x-mean))/

        return log_likelihood

    ## quetion ii
    def fit(self):
        """Function computes likelihood parameters from training data in the \
        training phase"""

        self.priors = np.divide(np.bincount(self.y_train), self.y_train.shape[0])

        # calculate per class data likelihoods
        if np.shape(self.priors)[0] == 3:
          self.feature_means = np.concatenate(([np.mean(self.X_train[self.y_train == 0], axis
          self.feature_std = np.concatenate(([np.std(self.X_train[self.y_train == 0], axis=0)

        if np.shape(self.priors)[0] == 2:
          self.feature_means = np.concatenate(([np.mean(self.X_train[self.y_train == 0], axis
          self.feature_std = np.concatenate(([np.std(self.X_train[self.y_train == 0], axis=0)

    ## quetion iii
    def predict(self, X_test):
        """Function computes the normalized posterior probabilities and class \
        predictions for the provided test data.

        Parameters:
        -----------
        X_test: ndarray of shape (N,D)
            2D numpy array containing N testing examples having D dimensions each.

        Returns:
        --------
        y_pred: ndarray of shape (N,1)
            vector containing class predictions for each of the N training\
             points in X_test.

        posteriors: ndarray of shape (N,C)
            numpy array containing normalized class posterior probabilities \
            for each of the C classes for each training example.

        """
        posteriors = np.zeros((np.shape(X_test)[0], np.shape(self.priors)[0]))



        for ii in range(np.shape(X_test)[0]):
          p = np.exp( np.sum( log_gaussian( np.reshape(X_test[ii, :], (1,-1)) , self.feature_
          posteriors[ii, :] = p / np.sum(p)

        y_pred = np.argmax(posteriors, axis = 1)

        return y_pred, posteriors

#-----------------Don't change anything below------------------------#

# define train and test sizes
N_train = 20
N_test = 150 - N_train
```

```
# load data
iris = load_iris()
X, y = iris.data, iris.target

# train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=N_test,
                                                    train_size=N_train,
                                                    random_state=4803)

model = MyNaiveBayes(X_train, y_train)
model.fit()
y_pred, _ = model.predict(X_test)

print("Accuracy Score: %.3f" % accuracy_score(y_test, y_pred))
```

```
     Accuracy Score: 0.938
```

Problem 2: Logistic Regression for Binary Classification (40pts)

Logistic regression is a popular machine learning algorithm for binary classification problems. Here, we will use the `breast_cancer` dataset in `sklearn` to guide you through building your own classifier. The target variable y can be modeled as a binary random variable taking on values in the set $[0, 1]$ via a bernoulli distribution characterized by the probability of success, p, conditioned on the $d-$dimensional feature vector $\mathbf{x} \in \mathbb{R}^d$. Additionally, p is obtained by taking the sigmoid of $\mathbf{x}$. This formulation is shown below:

$$\begin{aligned} P(y|\mathbf{x}) &= Ber(y; p) \\ &= Ber(y; \sigma(\mathbf{x})) \\ &= \sigma(\mathbf{x})^y \times (1 - \sigma(\mathbf{x}))^{1-y}, \qquad (1) \end{aligned}$$

where $\sigma(\mathbf{x}) = \frac{1}{1+e^{-w^T\mathbf{x}}}$ and $w \in \mathbb{R}^d$ is the parameter that we want to learn from dataset. You may notice: when $y = 0$, $P(y = 0|\mathbf{x}) = 1 - \sigma(\mathbf{x})$, and when $y = 1$, $P(y = 1|\mathbf{x}) = \sigma(\mathbf{x})$. It's the same as in the Lecture 4 page 19 with $b = 0$.

Problem 2 (a) Sigmoid Function (5pts)

Please complete the sigmoid function in the cell below. Calculate $P(y = 0|\mathbf{x})$ and $P(y = 1|\mathbf{x})$ with the given $w1$ and $w2$ separately and a test data feature x from `breast_cancer` dataset.

Solution to Problem 2 Part (a)

```
from operator import matmul
from sklearn.datasets import load_breast_cancer
cancer = load_breast_cancer()
x = cancer.data[0, :].reshape(1, -1)
y = cancer.target[0]

w1 = np.full(30, 0.002).reshape(-1, 1)
w2 = np.full(30, -0.002).reshape(-1, 1)

'''
P_y0x_w1: A float value
  Conditional probability of y=0 given x with w1.

P_y1x_w1: A float value
  Conditional probability of y=1 given x with w1.

P_y0x_w2: A float value
  Conditional probability of y=0 given x with w2.

P_y1x_w2: A float value
  Conditional probability of y=1 given x with w2.

'''

#-----------------Don't change anything above------------------------#

def sigmoid(X, w):
  """Computes sigmoid for given data array X and parameter w"""

  sigmoid_val = 1/ (1 + np.exp(-np.sum(np.matmul(X,w))))  ##TODO
  ## Note: np.exp( /Elementwise multiply w and X, sum that up and take a negative sign/ )

  return sigmoid_val

P_y0x_w1 = 1 - sigmoid(x, w1) ##TODO
P_y1x_w1 = sigmoid(x, w1) ##TODO

P_y0x_w2 = 1 - sigmoid(x, w2) ##TODO
P_y1x_w2 = sigmoid(x, w2) #TODO

#-----------------Don't change anything below------------------------#
print('feature x=', end=' ')
print(x)
print('label y=', end=' ')
print(y)

print('\nUsing w1:')
print('P(y=0|x)=', end=' ')
print(P_y0x_w1)

print('P(y=1|x)=', end=' ')
print(P_y1x_w1)

print('\nUsing w2:')
print('P(y=0|x)=', end=' ')
print(P_y0x_w2)

print('P(y=1|x)=', end=' ')
print(P_y1x_w2)
```

```
feature x= [[1.799e+01 1.038e+01 1.228e+02 1.001e+03 1.184e-01 2.776e-01 3.001e-01
  1.471e-01 2.419e-01 7.871e-02 1.095e+00 9.053e-01 8.589e+00 1.534e+02
  6.399e-03 4.904e-02 5.373e-02 1.587e-02 3.003e-02 6.193e-03 2.538e+01
  1.733e+01 1.846e+02 2.019e+03 1.622e-01 6.656e-01 7.119e-01 2.654e-01
  4.601e-01 1.189e-01]]
label y= 0

Using w1:
P(y=0|x)= 0.000798196732087253
P(y=1|x)= 0.9992018032679127

Using w2:
P(y=0|x)= 0.9992018032679127
P(y=1|x)= 0.0007981967320872286
```

Problem 2 (b) Logistic Regression Cost Function (10pts)

After we know the formulation of the logistic regression, we need to know how to train the parameter w using dataset. In part (a), we already see that different w can provide very different $P(y|\mathbf{x})$. Hence, our target here is to find the best w that maximizes $P(y|\mathbf{x})$ for the training data. As before, it is easier to work with logs of probabilities than the raw probabilities themselves, so we take the log on both sides of (1) to obtain:

$$\log P(y|x) = y \times \log \sigma(\mathbf{x}) + (1-y) \times \log\left(1-\sigma(\mathbf{x})\right)$$

The model training involves maximizing $\log P(y|x)$ over all possible values of w via an MLE formulation. The equivalent of this is to minimize the negative log-likelihood, $-\log P(y|x)$ over w. This optimization problem is shown below:

$$w^* = \underset{w}{\operatorname{argmin}} -y \times \log \sigma(\mathbf{x}) - (1-y) \times \log\left(1-\sigma(\mathbf{x})\right)$$

Since the training data usually consists of multiple labeled training examples, $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$, the optimization is carried out over the expected log likelihood loss, as shown below:

$$\begin{aligned} w^* &= \underset{w}{\operatorname{argmin}} \underset{(\mathbf{x},y)\backsim P(\mathbf{x},y)}{\mathbb{E}}\left[-y \times \log \sigma(\mathbf{x}) - (1-y) \times \log\left(1-\sigma(\mathbf{x})\right)\right] \\ &= \underset{w}{\operatorname{argmin}} \frac{1}{N}\sum_{i=1}^{N} -y_i \times \log \sigma(\mathbf{x}_i) - (1-y_i) \times \log\left(1-\sigma(\mathbf{x}_i)\right) = LL(\mathbf{x}, y, w) \qquad (2 \end{aligned}$$

Equation (2) is the cost function of logistic regression, and by minimizing this cost function with w, we can maximize the $P(y|\mathbf{x})$ for the training data.

Question: Complete the cost function in the cell below. Calculate the cost function value using the given w_1 and w_2 separately with the data from breast_cancer dataset. (You may need to use the sigmoid function in problem 2 (a)) Which w is better? w_1 or w_2?

Solution to Problem 2 Part (b)

```
cancer = load_breast_cancer()
data = cancer.data
label = cancer.target

w1 = np.full(30, 0.002).reshape(-1, 1)
w2 = np.full(30, -0.002).reshape(-1, 1)

'''
cost_1: A float value
  Cost function value with w1.

cost_2: A float value
  Cost function value with w2.

'''

#-----------------Don't change anything above------------------------#

def logit_cost_func(w, X, y):
  """function computes value of cost function given the w vector, the feature data, and corre
  """
  cost = np.mean( - np.multiply(   np.reshape(y, (-1,1)), np.log(1/(1+np.exp(-np.matmul(X, w)

  return cost

cost_1 = logit_cost_func(w1, data, label)

cost_2 = logit_cost_func(w2, data, label)

#-----------------Don't change anything below------------------------#
print('Cost function value with w1:', end=' ')
print(cost_1)
print('Cost function value with w2:', end=' ')
print(cost_2)
```

```
    Cost function value with w1: 2.16773861630246
```

```
Cost function value with w2: 1.6662547928342286
```

Solution to Problem 2 Part (b) continued:

w2 is better since it has a smaller cost value.

Problem 2 (c) Solve the Optimization Problem with Gradient Descent (10pts)

Now, our target is to find the w associated to the minimum of the cost function (2). You may remember from your calculus classes how the derivative is used to calculate the minima/maxima of a function. This is done by obtaining the expression for the derivative, setting it equal to zero, and then solving for the equation.

However, in the case of the logistic regression cost function, there is no closed form solution to the equation; rather the equation is solved via an iterative minimization algorithm called the **Gradient Descent**. Training may be stopped once the algorithm has sufficiently converged, as measured by either the amount of change happening to the cost function over successive iterations, or by prespecifying the number of iterations. The expression for the gradient of the logistic regression objective function is given below:

$$\frac{\partial LL(\mathbf{x}, y, w)}{\partial w_j} = -\frac{1}{N}\sum_{i=1}^{N}(y_i - \sigma(\mathbf{x}_i))x_{ji},$$

where w_j is component of the vector w and x_j the *j-th* component of the *i-th* training example, $\mathbf{x}_i$. Each gradient descent step performs the following update:

$$w_j^{k+1} = w_j^k - \text{step} \times \frac{\partial LL(\mathbf{x}, y, w)}{\partial w_j}, \quad k = 0, 1, \ldots, K$$

Question: Complete the gradient of cost function and gradient descent function in the below cell. Use the provided parameters to get the w^K.

Solution to Problem 2 Part (c)

```
cancer = load_breast_cancer()
data = cancer.data
label = cancer.target

w0 = np.full(30, 0).reshape(-1, 1) # initial w0
num_epochs = 100  ## K
```

```
step_size = 0.0001 ## step size

'''
w_K: ndarray of shape (D, 1)
  The parameter vector that the graident descent ends up with.

'''

#-----------------Don't change anything above------------------------#

def logit_grad(w, X, y):
  """Function computes the gradient of the logistic regression given the w vector,
  the data tensor , and corresponding targets"""

  sigmoid = 1/( 1 + np.exp(-np.matmul(X, w)) )

  grad = - np.matmul ( np.transpose(X), np.subtract( y.reshape((569,1)) , sigmoid ) )  / (np.

  return grad

def grdient_descent(w0, X, y, num_epochs=20, step=2):
  """Function performs gradient descent to compute optimal w.

  Parameters:
  -----------
  w0: ndarray of shape (D,),
      initial of parameter vector w.

  X: ndarray of shape (N, D),
      feature tensor of dataset.

  y: ndarray of shape (N,),
      label vector of dataset.

  num_epochs: int,
      integer specifying the number of training epochs for the gradient descent algorithm.

  step: float,
      float specifying the step size in the gradient descent algorithm.

  """
  w = w0
  for epoch in range(num_epochs):

    w = w - step * logit_grad(w, X, y)

  return w

w_K = grdient_descent(w0, data, label, num_epochs, step_size)

#-----------------Don't change anything below------------------------#
```

```
print('w^K =', end=' ')
print(w_K)

    w^K = [[ 1.40313887e-02]
     [ 2.60313493e-02]
     [ 8.58982139e-02]
     [ 1.23094663e-01]
     [ 1.46600927e-04]
     [ 4.96866080e-05]
     [-7.81343559e-05]
     [-3.99877194e-05]
     [ 2.76785712e-04]
     [ 1.09541487e-04]
     [ 9.18611777e-05]
     [ 2.08930997e-03]
     [ 5.39362358e-04]
     [-2.63569439e-02]
     [ 1.28138963e-05]
     [ 2.11292384e-05]
     [ 2.14649083e-05]
     [ 9.51989286e-06]
     [ 3.48895113e-05]
     [ 5.60291184e-06]
     [ 1.34450299e-02]
     [ 3.31080278e-02]
     [ 8.19111429e-02]
     [-9.38451396e-02]
     [ 1.91528800e-04]
     [ 5.73558378e-05]
     [-9.93683771e-05]
     [-1.55272368e-05]
     [ 3.98596043e-04]
     [ 1.22092229e-04]]
```

Problem 2 (d) Logistic Regression Classifier Implementation (15pts)

Finally, for the inference phase on the test data, the trained weights are used to compute posterior probabilities on test examples, which are then classified as belonging to either of the two classes depending on if the posterior is greater than or less than 0.5, as shown below:

$$P(y|\mathbf{x}_i^{test}) = \sigma(w^T \mathbf{x}_i^{test})$$

$$\hat{y} = \begin{cases} 1 & P(y|\mathbf{x}_i^{test}) > 0.5 \\ 0 & \text{otherwise} \end{cases}$$

In this exercise, you are going to implement your very own Logistic Regression class, with its `fit()` and `predict()` functions, among others. We will work with the `breast_cancer` dataset (accessed via `load_breast_cancer` function in `sklearn`) as an example, but the class should be able to take as input any other real valued feature data of any number of features and training examples, and be able to predict binary classes based on the MLE principle for unseen test data, as well as return the normalized posterior probabilities. Carefully read the questions below and answer them appropriately.

i) Complete the `sigmoid` function. This should be similar to problem 2 (a).

ii) Complete the `logit_cost_func` function. This should be similar to problem 2 (b).

ii) Complete the `logit_grad` function. This should be similar to problem 2 (c).

iv) Complete the `fit` function with gradient descent algorithm. This should be similar to problem 2 (c).

v) Implement the prediction routine elaborated above in the body of the `predict` function below. Finally, execute the code cell and describe what you observe.

vi) Assuming you implemented everything correctly, the algorithm should have worked and yet, it fails to perform decently. The reason for that is the un-normalized and un-scaled training and test data. Uncomment the line below performing the normalization and execute the code cell again. The performance should be much better. Interestingly, normalizing the data makes little to no difference to the performacne of the naive bayes classifier. Go back to Question 1 and verify this for yourselves. Why do you think normalization is so vital for Logistic Regression but hardly matters for Naïve Bayes?

> Do not change the function definitions for the functions defined in the MyLogisticRegression class template below. They should take inputs and output results of the form indicated. You are free to add other internal functions and use them inside the class definition as you see convenient. However, that should not change the external code's structure, nor the shape and form of the outputs returned. ***Note***: Any variable preceded by the `self.` keyword gets stored by the class structure and can be used and changed afterwards inside the class regardless of whether the function that first made it returns it or not.

Solution to Problem 2 Part (d)

```
from os import waitid
import numpy as np
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

import matplotlib.pyplot as plt
```

```
class MyLogisticRegression:
    def __init__(self, X, y):
        """Function intializes the Logistic Regression class.

        Parameters:
        -----------
        X_train: ndarray of shape (N,D)
            Numpy array containing N training examples, each D dimensional.

        y_train: ndarray of shape (N,)
            Numpy array containing vector of ground truth classes for examples in X_train
        """

        self.X = np.concatenate((X, np.ones((X.shape[0], 1))), axis=1)
        self.y = y
        self.w = np.random.randn(self.X.shape[1],1)

    ## question i
    def sigmoid(self, X, w):
        """Computes sigmoid for given data array X"""

        sigmoid_val = 1 / ( 1 + np.exp( -np.matmul(X, w) ) )

        return sigmoid_val

    ## question ii
    # define logistic regression cost function
    def logit_cost_func(self, w, X, y):
        """function computes value of cost function given the w vector, the feature tensor, a
        """

        cost = np.mean( - np.multiply(   np.reshape(y, (-1,1)), np.log(1/(1+np.exp(-np.matmul

        return cost

    ## question iii
    # define gradient function
    def logit_grad(self, w, X, y):
        """Function computes the gradient of the logistic regression given the w vector,
        the tensor , and corresponding targets"""

        sigmoid = 1/( 1 + np.exp(-np.matmul(X, w)) )

        grad = - np.matmul ( np.transpose(X), np.subtract( y.reshape((y.shape[0],1)) , sigmoi

        return grad

    ## question iv
    def fit(self, num_epochs=20, step=2):
        """Function performs gradient descent to compute optimal w.
        Parameters:
        -----------
        num_epochs: int,
            integer specifying the number of training epochs for the gradient descent algorit

        step: float,
            float specifying the step size in the gradient descent algorithm.

        """

        w = self.w

        for epoch in range(num_epochs):
          w = w - step * self.logit_grad(w, self.X, self.y)

        self.w = w

    ## question v
    def predict(self, X):
        """Function computes the normalized posterior probabilities and class predictions for

        Parameters:
        -----------
        X: ndarray of shape (N,D)
            2D numpy array containing N testing examples having D dimensions each.

        Returns:
        --------
        y_pred: ndarray of shape (N,1)
            vector containing class predictions for each of the N training points in X_test.

        probs: ndarray of shape (N,1)
            numpy array containing normalized class posterior probabilities for each of the p

        """
        X = np.concatenate((X, np.ones((X.shape[0], 1))), axis=1)

        probs = self.sigmoid(X, self.w)
        ##TODO # write code to compute posterior probabilities on test data

        y_pred = np.zeros(np.shape(probs))
        y_pred[probs > 0.5] = 1

        return y_pred, probs

#-----------------Don't change anything below------------------------#

# define train and test sizes
N_train = 20
N_test = 150 - N_train
```

```
# load data
cancer = load_breast_cancer()
X, y = cancer.data, cancer.target

# Normalize X. Only uncomment for part e
X = (X - np.min(X, axis=0, keepdims=True)) / (np.max(X, axis=0, keepdims=True) - np.min(X, a

# train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=N_test, train_size=N_trai

# train model and predict on test data
model =  MyLogisticRegression(X_train, y_train)
model.fit(num_epochs = 50, step=2)
y_pred, probs = model.predict(X_test)

# compute accuracy
print("Accuracy Score: %.2f" % accuracy_score(y_pred, y_test))
```

```
Accuracy Score: 0.91
```

Solution to Problem 2 Part (d) continued:

vi)

Normalization is done in both Problem 1 and Problem. However, the accuray of Problem 2 (Logistic Regression) increases more, approximately from ~0.66 to ~0.9, while the accuracy of Problem 1 (Naive Bayes) remains the same.

This is most likely due to the different calculation processes for two classifiers:

- For Naive Bayes, the normalization process will only affect the `posterior` since the porcess is `posterior / np.sum(posterior)`. That said, all data are scaled at the same degree so the effects on individuals are small. Additionally, the critical factor that affects the accracy will be the likelihood, which depends on the calculated mean and standard diviation from the features' actual values, instead of the posterior. Therefore, the accuracy will not change much here.
- For Logistic Regression, the gradient desent process of finding the cost of determines the accuracy. The process of normalization will help stablize the gradient desent and will thus affect the accuracy a lot.

Problem 3: Performance Evaluation Metrics for Classification Accuracy (20pts)

Note: it may take another week before we cover, in lecture, the material that covers this problem.

Having just implemented two classifiers and done an initial performance evaluation via the accuracy score, we now move onto a more thorough and involved performance analysis. Although the accuracy metric may provide a quick baseline to judge results with, it can be very misleading in case of imbalanced training datasets, where some classes are dominated by others. Another performance metric that can be very useful in this case is the **Receiver Operating Characteristics (ROC) curve**. It is essentially a plot of the True Positive Rate (TPR) against the False Positive Rate (FPR) for various thresholds on the posterior probabilties predicted by the classifier. Read the following questions carefully and answer as appropriate.

Problem 3 (a) (10pts)
In the template code given below, fill in code to compute the TPRs and FPRs for a given posterior probability vector and the corresponding vector of ground-truths.

Problem 3 (b) (5pts)
Execute the code to generate a plot of ROC curves for both the logistic regression classifier and the Naïve Bayes classifier for different training set sizes on the `breast_cancer` dataset. What do you observe in each individual subplot regarding how the accuracy metric and the roc curve change relative to the training set size? What do you think is the explanation for this trend? Now compare the the individual plots to each other. Which classifier is more robust in the case of limited data on the `breast_cancer` dataset?

Problem 3 (c) (5pts)
We are given four classifiers for which we observe the following cases:

i. Low TPR, Low FPR
ii. Low TPR, High FPR
iii. High TPR, Low FPR
iv. High TPR, High FPR

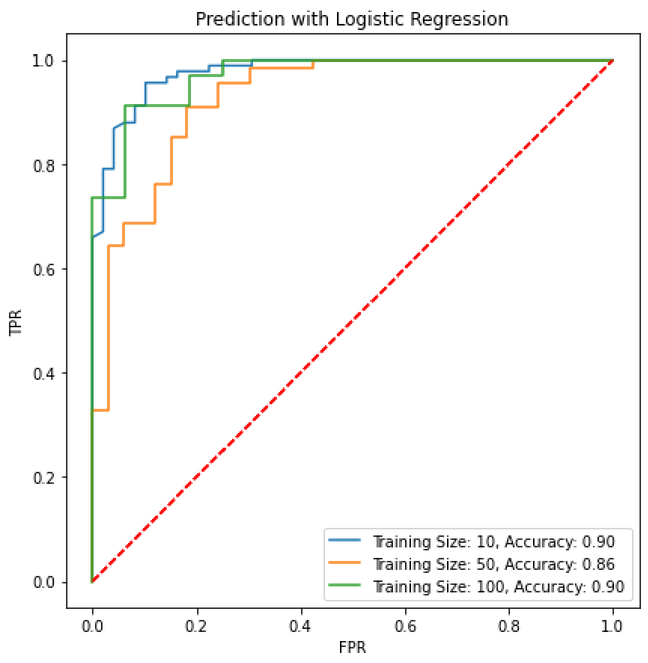
What does each situation tell us about the respective classifier? Explain in terms of what you think the training data distribution might have been and/or the particular predictive nature of the classifier.

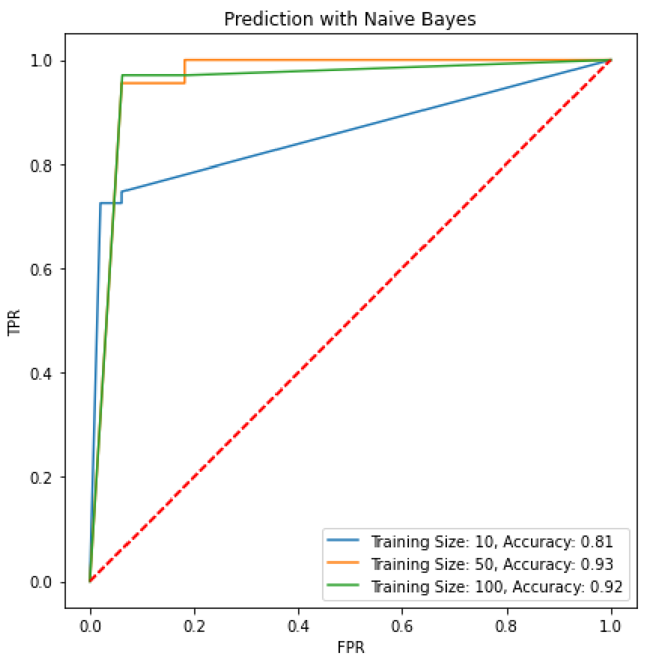
Solution to Problem 3 Part (a):

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
```

```
def roc(probs, y_test):
    """Function returns TPR and FPR given a vector of probabilities and another
    for ground-truth predictions

    Parameters:
    -----------
    probs: ndarray of shape (N,1)
        numpy array of normalized posterior porbabilities for each of N examples in a test da

    y_test: ndarray of shape (N,1)
        numpy array containing ground-truth class predictions for each of the N examples in t

    Returns:
    --------
    TPR: ndarray of shape (D,1)
        numpy array containing true positive rates for each of the D thresholds applied to th

    FPR: ndarray of shape (D,1)
        numpy array containing false positive rates for each of the D thresholds applied to t
        """
    truely_positive = np.sum(y_test==1)
    truely_negative = np.sum(y_test==0)

    th = np.linspace(-0.1, 1, 1000)

    TPR = np.zeros(np.shape(th)[0])
    FPR = np.zeros(np.shape(th)[0])

    for ii in range(np.shape(th)[0]):
      TPR[ii] = np.sum(np.reshape(np.array(y_test==1), [-1,1]) & np.array(probs > th[ii]))/ t
      FPR[ii] = np.sum(np.reshape(np.array(y_test==0), [-1,1]) & np.array(probs > th[ii]))/ t

    return TPR, FPR

#


#-----------------Don't change anything below------------------------#

# load data
cancer = load_breast_cancer()
X, y = cancer.data, cancer.target

# Normalize X
X = (X - np.min(X, axis=0, keepdims=True)) / (np.max(X, axis=0, keepdims=True) - np.min(X, ax

# define training set sizes
training_set_sizes = [10, 50, 100]
```

```
    # set up plots
    fig, (ax1, ax2) = plt.subplots(1,2, figsize=(15,7))

    for train_size in training_set_sizes:

        # define train and test sizes
        N_train = train_size
        N_test = 150 - N_train

        # train-test split
        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=N_test, train_size=N_

        # train model and predict on test data
        model =  MyLogisticRegression(X_train, y_train)
        model.fit(num_epochs = 50, step=2)
        y_pred, probs = model.predict(X_test)


        # plot ROC for logistic regression
        TPR, FPR = roc(probs, y_test)
        ax1.plot(FPR, TPR, label='Training Size: %d, Accuracy: %.2f'%(train_size, accuracy_score(
        ax1.plot([0,1],[0,1], linestyle='--', color='red')
        ax1.set_xlabel('FPR')
        ax1.set_ylabel('TPR')


        # train model and predict on test data
        model =  MyNaiveBayes(X_train, y_train)
        model.fit()
        y_pred, probs = model.predict(X_test)

        # plot ROC for logistic regression
        TPR, FPR = roc(probs[:,1].reshape(-1,1), y_test)
        ax2.plot(FPR, TPR, label='Training Size: %d, Accuracy: %.2f'%(train_size, accuracy_score(
        ax2.plot([0,1],[0,1], linestyle='--', color='red')
        ax2.set_xlabel('FPR')
        ax2.set_ylabel('TPR')



    ax1.set_title('Prediction with Logistic Regression')
    ax2.set_title('Prediction with Naive Bayes')
    ax1.legend()
    ax2.legend()
    plt.show()
```





Solution to Problem 3 Part (b):

- From both classifiers, we can see that the accuracy increases as the training size gets larger.
    - Especially while using the Naive Bayes classifier, the accuracy improve more obviously (from 0.81 to 0.92) comparing to Logistic Regression (from 0.87 to 0.9)
- Theoretically, a larger data set should also help us make a more accurate estimation statistically when doing any experimentations or study.
    - Moreover, since probability distribution depends on sampling and statistics a lot, the gaussian distribution we used in Naive Bayes would represent the actual data much better. This could explain why Naive Bayes have a larger increase in accuracy.

Solution to Problem 3 Part (c):

Generally, TPR defines how many correct positive results occur among all positive samples available during the test. FPR, defines how many incorrect positive results occur among all negative samples available during the test. Based on these two facts:

1. Low TPR, Low FPR:
    - In this case, both the results of true positive and false positive are very few comparing to the other 2 categories. That said, **there is a high chance to get a negative for the result no matter what's the actual value**.
        - These data points lies at the bottom left corner of the plot.
2. Low TPR, High FPR

- In this case, there are more false positive and false negative comparing to the other 2 categories. That is, the false predicted data is the majority of the result. This indicates that the the classifier is inaccurate and is doing a really bad job of predicting the actual data.
- This also shows that the training data represents the real world poorly. **These models should be avoided.**
  - This is not shown in the plot.

3. High TPR, Low FPR
   - In this case, there are more true positive and true negative comparing to the other 2 categories. That is, the true predicted data is the majority of the result. This indicates that the the classifier is very percise and is doing a great job of predicting the actual data.
   - This also shows that the training data represents the real world well. **The model with High TPR and Low FPR is what we prefered.**
     - These data points lies at the top left side of the plot.
4. High TPR, High FPR
   - In this case, there are more results of true positive and false positive comparing to the other 2 categories. That is, the true predicted data is the majority of the result. That said, **there is a high chance to get a positive for the result no matter what's the actual value**.
     - These data points lies at the top right side of the plot.

✓ 0s completed at 3:59 PM