Georgia Institute of Technology

ECE 4252/8803: Fundamentals of Machine Learning (FunML)

Spring 2025

Homework Assignment # 2

Due: January 31, 2025 @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.
- 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.
- Print a PDF copy of the notebook with all its outputs printed. Zip both **PDF** and **IPYNB** in a single **ZIP** file and submit it on `Canvas` under Assignments.
- Rename the PDF, IPYNB and ZIP file according to the format:
  LastName_FirstName_ECE_4252_8803_F24_assignment_2.zip
  LastName_FirstName_ECE_4252_8803_F24_assignment_2.pdf
  LastName_FirstName_ECE_4252_8803_F24_assignment_2.ipynb
- 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 the first lecture 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:

Remeber to convert your homework into PDF format before submitting it.
Here is a video summarizing how to export ipython Notebook into PDF.

- **[Method 1: Print to PDF]**
  After you run every cell and get their outputs, you can use **[File] -> [Print Preview]** , and then use your browser's print function. 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.
- **[Method 2: GoFullPage Chrome Extension]**
  Install the extension and generate PDF file of the Ipython Notebook in the browser.

Note: Since we have embedded images in HW1, it's recommended to generate PDF using the first method. Also, Georgia Tech provides a student discount for Adobe Acrobat subscription. Further information can be found here.

What is a good Strategy for FunML Homework?

- Understand the algorithm → Implement Python code step by step → Verify the result.
  (Every step is important, and the order is also important.)

- Understand the concepts and math of the algorithm.
  - Slides
  - Reading, References, & Resources
  - Homework descriptions

- Prcatice Numpy techniques and skills.
  - indexing
  - dimension/broadcasting
  - mask/boolean array

Problem 1: Naïve Bayes for Classifying Real-Valued Data (35pts)

In Lecture 2, 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 --user scikit-learn
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'])
```

```
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.11/dist-packages (1.6.1)
Requirement already satisfied: numpy>=1.19.5 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (1.13.1)
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn) (3.5.0)
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 = $P(x_1 \mid y = c_1)$

Marginal Likelihood = $P(x_1)$

Prior Probabilities = $P(y = c_1)$

Posterior Probabilities = $P(y = c_1|x_1)$

$P(x_1 \mid y = c_1)$ expresses the probability of observing a specific feature x_1, given that the flower belongs to the class c_1.

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]

Hint: You may check different functions of NumPy to count the occurrences of each value in an array.

Solution to Problem 1 Part (b)

$P(y=0)$	$P(y=1)$	$P(y=2)$
0.333	0.333	0.333

```
labels = iris.target

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

N_ck: ndarray of shape (3,)
  Number of examples in each class.

P_yck: ndarray of shape (3,)
  Priori of each class.
'''

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

N = len(iris.data)

N_ck = np.bincount(iris.target)

P_yck = 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 (5pts)

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)

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

	$\hat{\sigma}_{1y}$	$\hat{\sigma}_{2y}$	$\hat{\sigma}_{3y}$	$\hat{\sigma}_{4y}$
$y=0$	6.53333333	3.06666667	5.60000000	1.96666667
$y=1$	0.57154761	0.28284271	0.43204938	0.18856181
$y=2$	0.26246693	0.12472191	0.08164966	0.23570226

```
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-----------------------#

for i in range(3):
  mu[i] = np.mean(features[labels == i], axis=0)
  sigma[i] = np.std(features[labels == i], axis=0)

#-----------------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:

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

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:

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

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, $P(x_j | y = c_k) = \mathcal{N}(\mu_{jc_k}, \sigma^2_{jc_k})$. 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)

1. $\log P(x_j | y = c_k) = -\frac{1}{2}\log(\pi\sigma^2_{jc_k}) - \frac{(x_j - \mu_{jc_k})^2}{2\sigma_{jc_k}}$
2. Answered in the code cell below
3. The test example belongs to setosa.

```
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 =  -0.5 * np.log(np.pi * std**2) - (x - mean)**2 / (2 * std**2)

  return log_likelihood

## question ii
posteriors = np.zeros(3)
for i in range(3):
  log_likelihood = 0
  for j in range(4):
```

```
      log_likelihood += log_gaussian(x_test[0][j], mu[i][j], sigma[i][j])
    posteriors[i] = log_likelihood + np.log(P_yck[i])

posteriors = np.exp(posteriors)
posteriors /= sum(posteriors)

## question iii
y_hat = np.argmax(posteriors)

#-----------------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

    ## question 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 =  -0.5 * np.log(np.pi * std**2) - (x - mean)**2 / (2 * std**2)

            return log_likelihood

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

            n_classes = len(np.unique(self.y_train))
            n_features = self.X_train.shape[1]
            self.priors = np.zeros(n_classes)

            for i in range(n_classes):
                self.priors[i] = np.mean(self.y_train == i)

            self.feature_means = np.zeros((self.priors.size, self.X_train.shape[1]))
            self.feature_std = np.zeros((self.priors.size, self.X_train.shape[1]))
            for i in range(n_classes):
                self.feature_means[i] = np.mean(self.X_train[self.y_train == i], axis=0)
                self.feature_std[i] = np.std(self.X_train[self.y_train == i], axis=0)
            #log likelihoods were calculated in iii

        ## question 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.

            """

            log_posteriors = np.zeros((X_test.shape[0], self.priors.size))
            for i in range(X_test.shape[0]):
                for j in range(self.priors.size):
                    log_likelihood = 0
                    for k in range(self.X_train.shape[1]):
                      log_likelihood += self.log_gaussian(X_test[i][k], self.feature_means[j][k], self.feature_std[j][k])
                    log_posteriors[i][j] = log_likelihood + np.log(self.priors[j])
            # perform inference
            posteriors = np.exp(log_posteriors)
            posteriors /= np.sum(posteriors, axis=1, keepdims=True)
            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 (35pts)

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 3 page 27 with $b = 0$.

Problem 2 (a) Sigmoid Function (3pts)

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 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.dot(X, w)))

  return sigmoid_val

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

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

#-----------------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)= [[0.0007982]]
P(y=1|x)= [[0.9992018]]

Using w2:
P(y=0|x)= [[0.9992018]]
P(y=1|x)= [[0.0007982]]
```

Problem 2 (b) Logistic Regression Cost Function (4pts)

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 corresponding labels
      """

      cost = -np.mean(y * np.log(sigmoid(X, w)) + (1 - y) * np.log(1 - sigmoid(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: 1.4438427525876945
Cost function value with w2: 2.3901506565483066
```

[Put your answer in this cell]

Problem 2 (c) Solve the Optimization Problem with Gradient Descent (3pts)

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"""

  grad = -np.mean((y.reshape(-1,1) - sigmoid(X, w)) * X, axis=0).reshape(-1,1)

  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 = [[ 1.40313714e-02]
 [ 2.60313351e-02]
 [ 8.58981078e-02]
 [ 1.23094523e-01]
 [ 1.46600786e-04]
 [ 4.96865930e-05]
 [-7.81343956e-05]
 [-3.99877235e-05]
 [ 2.76785560e-04]
 [ 1.09541419e-04]
 [ 9.18615082e-05]
 [ 2.08930875e-03]
 [ 5.39365876e-04]
 [-2.63567931e-02]
 [ 1.28138873e-05]
 [ 2.11292153e-05]
 [ 2.14648965e-05]
 [ 9.51987801e-06]
 [ 3.48894881e-05]
 [ 5.60290574e-06]
 [ 1.34450111e-02]
 [ 3.31080070e-02]
 [ 8.19110281e-02]
 [-9.38450272e-02]
 [ 1.91528645e-04]
 [ 5.73557155e-05]
 [-9.93683636e-05]
 [-1.55273115e-05]
 [ 3.98595810e-04]
 [ 1.22092134e-04]]
```

Problem 2 (d) Logistic Regression Classifier Implementation (13pts)

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 performance 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.

Note 1: Until this point, we have ignored the bias term b. However, it palys a crucial role in the initial rounds of training. A computationally attractive way of defining bias is by concatenating all ones to the input and letting gradient descent learn it via an additional column in w. The concatenation step is performed for you below. More on this in Lecture 7.

Note 2: Evaluation terms including ROC, TPR, and FPR will be covered in Lecture 6.

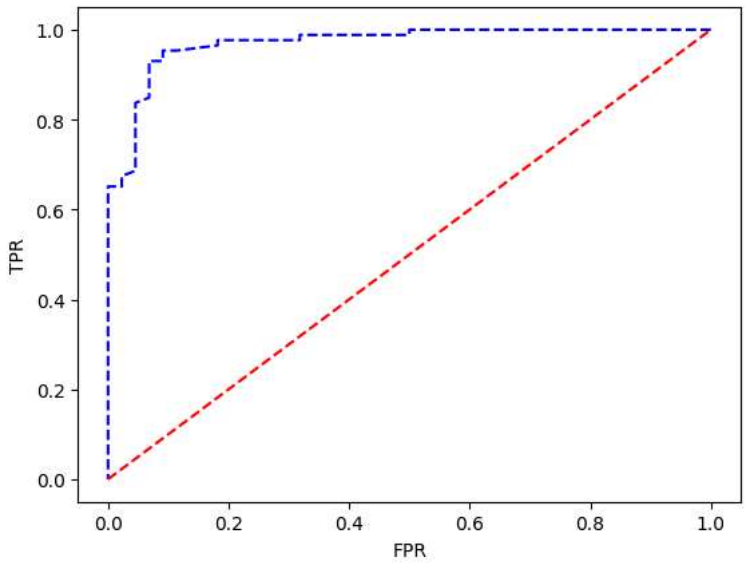
Solution to Problem 2 Part (d)

```
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
        """
        #The concatenation line below is a common trick to implicitly implement the bias term. We will learn more about this in Lecture 7.
        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.dot(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, and corresponding targets
            """
            predictions = self.sigmoid(X, w)
            cost = -np.mean(y * np.log(predictions) + (1 - y) * np.log(1 - predictions))

            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"""

            grad = -np.mean((y.reshape(-1,1) - self.sigmoid(X, w)) * X, axis=0).reshape(-1,1)

            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 algorithm

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

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

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

            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 positive class

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

            probs = self.sigmoid(X, self.w)
            y_pred = (probs > 0.5).astype(int)

            return y_pred, probs

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

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

        thresholds = np.linspace(0,1,100)
        mask = (probs > thresholds)*1
        TPR = np.sum(mask * y_test.reshape(-1,1), axis=0) / y_test.sum()
```

```
    FPR = np.sum(mask * (1-y_test.reshape(-1,1)), axis=0) / (1 - y_test).sum()

    return TPR, FPR


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

# 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)

# 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))
print("\nTrue Positive Rate: %.2f" % (np.sum(y_pred.reshape(-1,1) * y_test.reshape(-1,1))/np.sum(y_test)))
print("False Positive Rate: %.2f" % (np.sum(y_pred.reshape(-1,1) * (1 - y_test).reshape(-1,1))/np.sum(1 - y_test)))

# plot ROC
TPR, FPR = roc(probs, y_test)
plt.plot(FPR, TPR, linestyle='--', color='blue')
plt.plot([0,1],[0,1], linestyle='--', color='red')
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.show()
```

```
Accuracy Score: 0.92

True Positive Rate: 0.98
False Positive Rate: 0.18
```



V. There is no normalization results in the blue line which is supported by the low accuracy score of way under 50%.

VI. Normalization does not have that much importance for Naive Bayes since all the input parameters are considered indepedent of each other. Logistic regression is more dependent on such nomralization due to gradient updates.

Problem 2 (e) Effect of Hinge Loss (4pts)

The logistic loss is not the only cost function that allows us to classify binary data. In this part we test the same algorithm, but we'll optimize the hinge cost function seen in class.

Hinge predictions:

$$\hat{y} = \begin{cases} 1 & \mathbf{w}^T\mathbf{x} > 0 \\ 0 & \mathbf{w}^T\mathbf{x} <= 0 \end{cases}$$
$$L(\mathbf{x}, y\theta) = max(0, -y\mathbf{w}^T\mathbf{x})$$

| In this exercise, you are going to implement your very own hinge classifier 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 `hinge_cost_func` function. This should be similar to problem 2 (d).

ii) Complete the `hinge_grad` function. This should be similar to problem 2 (d).

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

iv) Implement the prediction routine elaborated above in the body of the `predict` function below. Note that the hinge classifier does not produce probabilities and make sure to return your predictions as 0 or 1. Y is scaled to -1 or 1 during the initialization of the class. Finally, execute the code cell and describe what you observe.

> Do not change the function definitions for the functions defined in the MyHingeClassifier 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.

How does this cost perform in comparison to the logistic cost seen before? Reflect on its advantages and disadvantages.

Solution to Problem 2 Part (e)

```
import numpy as np
from numpy.linalg import norm
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 MyHingeClassifier:
    def __init__(self, X, y):
        """Function intializes the Hinge Classifier 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 * 2 - 1).astype(int)
        self.w  = np.random.randn(self.X.shape[1],1)

    ## question i
    # define hinge loss cost function
    def hinge_cost_func(self, w, X, y):
        """function computes value of cost function given the w vector, the feature tensor, and corresponding targets
        """
        cost = np.mean(np.maximum(0, -y * np.dot(X, w)))

        return cost

    ## question ii
    # define gradient function
    def hinge_grad(self, w, X, y):
        """Function computes the gradient of the hinge loss given the w vector,
        the tensor , and corresponding targets"""
        y_pred = np.dot(X, w)
```

```
            margin = y_pred * y.reshape(-1,1)
            grad = -np.mean(y.reshape(-1,1) * X* (margin < 1), axis=0).reshape(-1,1)

            return grad

        ## question iii
        def fit(self, num_epochs=20, step=2):
            for epoch in range(num_epochs):
                print(f'Loss after epoch: {epoch}: ' \
                      f'{np.round(self.hinge_cost_func(self.w, self.X, self.y), 2)}')
                self.w = self.w - step * self.hinge_grad(self.w, self.X, self.y)

        ## question iv
        def predict(self, X):
            """Function computes the normalized posterior probabilities and class predictions for the provided test data.

            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 f
                or 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 positive class

            """
            X = np.concatenate((X, np.ones((X.shape[0], 1))), axis=1)
            y_pred = np.dot(X, self.w) > 0
            y_pred = y_pred.astype(int)
            return y_pred

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

    # 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)

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

    # compute accuracy
    print("Accuracy Score: %.2f" % accuracy_score(y_pred, y_test))
    print("\nTrue Positive Rate: %.2f" % (np.sum(y_pred.reshape(-1,1) * y_test.reshape(-1,1))/np.sum(y_test)))
    print("False Positive Rate: %.2f" % (np.sum(y_pred.reshape(-1,1) * (1 - y_test).reshape(-1,1))/np.sum(1 - y_test)))
```

```
Loss after epoch: 0: 0.2
Loss after epoch: 1: 0.42
Loss after epoch: 2: 0.96
Loss after epoch: 3: 1.05
Loss after epoch: 4: 0.63
Loss after epoch: 5: 1.46
Loss after epoch: 6: 0.59
Loss after epoch: 7: 1.03
Loss after epoch: 8: 0.79
Loss after epoch: 9: 1.43
Loss after epoch: 10: 0.87
Loss after epoch: 11: 1.05
Loss after epoch: 12: 0.94
Loss after epoch: 13: 1.11
Loss after epoch: 14: 1.01
Loss after epoch: 15: 1.18
```

```
Loss after epoch: 16: 1.05
Loss after epoch: 17: 1.14
Loss after epoch: 18: 1.06
Loss after epoch: 19: 1.18
Loss after epoch: 20: 1.07
Loss after epoch: 21: 1.17
Loss after epoch: 22: 1.12
Loss after epoch: 23: 1.24
Loss after epoch: 24: 1.1
Loss after epoch: 25: 1.22
Loss after epoch: 26: 1.14
Loss after epoch: 27: 1.26
Loss after epoch: 28: 1.13
Loss after epoch: 29: 1.25
Loss after epoch: 30: 1.2
Loss after epoch: 31: 1.27
Loss after epoch: 32: 1.19
Loss after epoch: 33: 1.31
Loss after epoch: 34: 1.18
Loss after epoch: 35: 1.3
Loss after epoch: 36: 1.27
Loss after epoch: 37: 1.27
Loss after epoch: 38: 1.27
Loss after epoch: 39: 1.27
Loss after epoch: 40: 1.27
Loss after epoch: 41: 1.27
Loss after epoch: 42: 1.27
Loss after epoch: 43: 1.27
Loss after epoch: 44: 1.27
Loss after epoch: 45: 1.27
Loss after epoch: 46: 1.27
Loss after epoch: 47: 1.27
Loss after epoch: 48: 1.27
Loss after epoch: 49: 1.27
Accuracy Score: 0.88

True Positive Rate: 0.93
False Positive Rate: 0.20
```

IV. The loss changes a little each time but expresses a strong model. The cost is higher than before with similar accuracy to the prior model.

Problem 2 (f) Effect of Learning Rate (3pts)

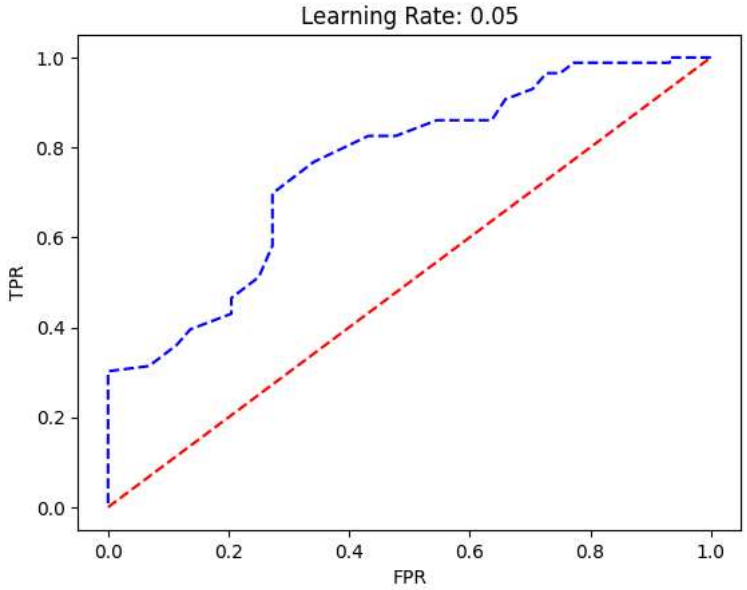
In this part, you will observe the effect of the learning rate on the algorithm's performance. Up until this part, you have used the step parameter as 2. Now, you will experiment with a few values to see the effect of the learning rate. Input the learning rates as 0.05, 2, and 100, respectively. Which one of these values produces a classifer with the highest accuracy? Discuss the issues by the other two learning rates.

Solution to Problem 2 Part (f)

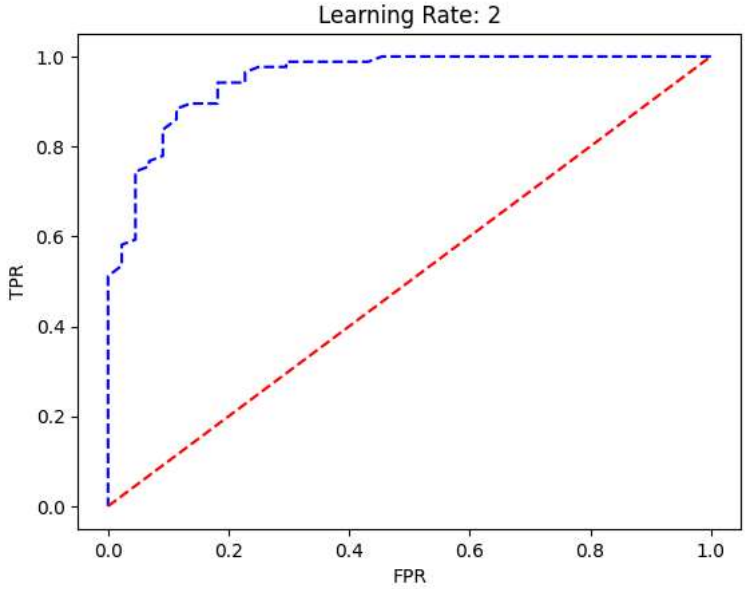
```
import numpy as np
from numpy.linalg import norm
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

import matplotlib.pyplot as plt



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

# 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)
```

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

learning_rates = 0.05, 2, 100

#-----------------Don't change anything below-----------------------#
for curr_lr in learning_rates:
  # train model and predict on test data
  model =  MyLogisticRegression(X_train, y_train)
  model.fit(num_epochs = 50, step=curr_lr)
  y_pred, probs = model.predict(X_test)

  # compute accuracy
  print("Accuracy Score: %.2f" % accuracy_score(y_pred, y_test))
  print("\nTrue Positive Rate: %.2f" % (np.sum(y_pred.reshape(-1,1) * y_test.reshape(-1,1))/np.sum(y_test)))
  print("False Positive Rate: %.2f" % (np.sum(y_pred.reshape(-1,1) * (1 - y_test).reshape(-1,1))/np.sum(1 - y_test)))

  # plot ROC
  TPR, FPR = roc(probs, y_test)
  plt.plot(FPR, TPR, linestyle='--', color='blue')
  plt.plot([0,1],[0,1], linestyle='--', color='red')
  title_str = "Learning Rate: " + str(curr_lr)
  plt.title(title_str)
  plt.xlabel('FPR')
  plt.ylabel('TPR')
  plt.show()
```

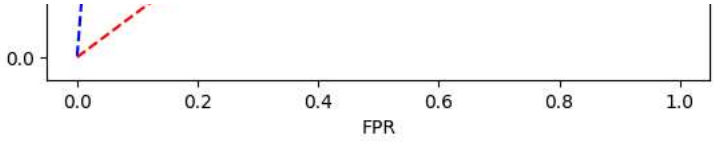
```
Accuracy Score: 0.52

True Positive Rate: 0.34
False Positive Rate: 0.11
```




```
Accuracy Score: 0.90

True Positive Rate: 0.94
False Positive Rate: 0.18
```




```
Accuracy Score: 0.87

True Positive Rate: 0.83
False Positive Rate: 0.05
```



There were better accuracy scores associated with higher learning rates with 0.05 learning too slowly and 100 taking too large of steps but learned the fastest.

Problem 2 (g) L2 Regularization (5 pts)

A challenge in machine learning is to create an algorithm that is generlizable, i.e., works for data beyond the training data. Several strategies that are used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. A subset of these strategies are known collectively as regularization techniques.

One of the simplest and most common technicques to induce parameter norm penalty is the L_2 parameter norm penalty, known as weight decay. By adding L_2 normalization to our cost function, from above, we get the following:

$$LL_{Regularized}(\mathbf{x}, y, w) = LL(\mathbf{x}, y, w) + \alpha * (w^T \cdot w)$$

$$\frac{\partial LL_{Regularized}(\mathbf{x}, y, w)}{\partial w_j} = \frac{\partial LL(\mathbf{x}, y, w)}{\partial w_j} + 2\alpha w_j$$

i) Change the `logit_cost_func` function to include the L_2 regularization.

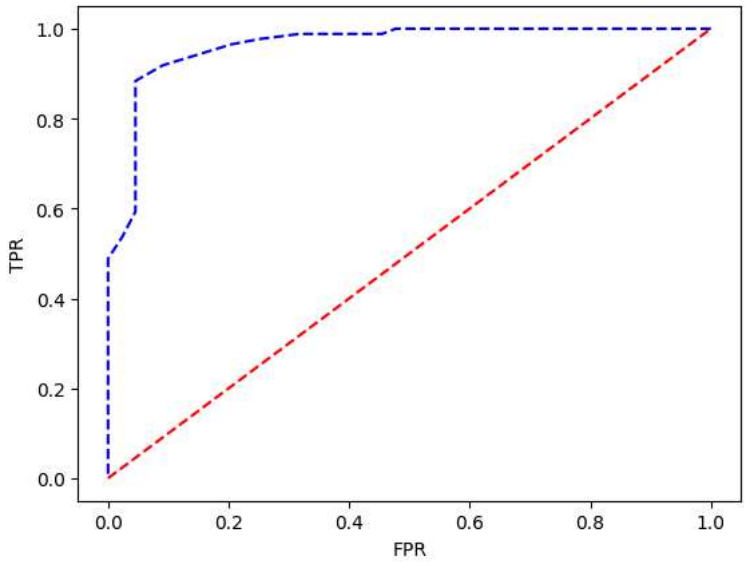
ii) Change the `logit_grad` function according to the cost function of L_2 regularization.

Solution to Problem 2 Part (g)

```
from copy import deepcopy

class MyLogisticRegressionWithRegularization(MyLogisticRegression):
    def __init__(self, X, y, alpha):
        super().__init__(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

        alpha: float
            The regularization constant
        """

        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)
        self.alpha = alpha

    ## question i
    # define logistic regression cost function with L2 regularization
    def logit_cost_func(self, w, X, y):
        """function computes value of cost function with L2 regularization given the w vector, the feature tensor, and corresponding targets
        """
        cost = -np.mean(y * np.log(self.sigmoid(X, w)) + (1 - y) * np.log(1 - self.sigmoid(X, w))) + self.alpha * np.dot(w.T, w)
        regression = self.alpha * np.dot(w.T, w)
        cost += regression
        return cost

    ## question ii
    # define gradient function with the L2 regularization term
    def logit_grad(self, w, X, y):
        """Function computes the gradient of the logistic regression with L2 regularization given the w vector,
        the tensor , and corresponding targets"""
        grad = -np.mean((y.reshape(-1,1) - self.sigmoid(X, w)) * X, axis=0).reshape(-1,1) + 2 * self.alpha * w
        regression = 2 * self.alpha * self.w
        grad += regression
```

```
        return grad

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

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

# define the range for the regularization parameter
alpha_range = np.arange(0,0.2,0.001)

# 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, axis=0, keepdims=True))

# 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)

# train model and predict on test data
best_model = []
best_accuracy = 0
for curr_alpha in alpha_range:
    model =  MyLogisticRegressionWithRegularization(X_train, y_train, alpha=curr_alpha)
    model.fit(num_epochs = 50, step=2)
    y_pred, probs = model.predict(X_test)

    if accuracy_score(y_pred, y_test) > best_accuracy:
        best_accuracy = accuracy_score(y_pred, y_test)
        best_model = deepcopy(model)

        # compute accuracy
        print("The regularization coefficient: %.3f" % curr_alpha)
        print("Accuracy Score: %.2f" % accuracy_score(y_pred, y_test))
        print("True Positive Rate: %.2f" % (np.sum(y_pred.reshape(-1,1) * y_test.reshape(-1,1))/np.sum(y_test)))
        print("False Positive Rate: %.2f" % (np.sum(y_pred.reshape(-1,1) * (1 - y_test).reshape(-1,1))/np.sum(1 - y_test)))
        print("\n")

# plot ROC
y_pred, probs = best_model.predict(X_test)
TPR, FPR = roc(probs, y_test)
plt.plot(FPR, TPR, linestyle='--', color='blue')
plt.plot([0,1],[0,1], linestyle='--', color='red')
plt.xlabel('FPR')
plt.ylabel('TPR')
plt.show()

print("The best regularization coefficient: %.3f" % curr_alpha)
print("Accuracy Score: %.2f" % accuracy_score(y_pred, y_test))
print("True Positive Rate: %.2f" % (np.sum(y_pred.reshape(-1,1) * y_test.reshape(-1,1))/np.sum(y_test)))
print("False Positive Rate: %.2f" % (np.sum(y_pred.reshape(-1,1) * (1 - y_test).reshape(-1,1))/np.sum(1 - y_test)))
print("\n")
```

```
The regularization coefficient: 0.000
Accuracy Score: 0.89
True Positive Rate: 0.92
False Positive Rate: 0.16


The regularization coefficient: 0.001
Accuracy Score: 0.91
True Positive Rate: 0.97
False Positive Rate: 0.20


The regularization coefficient: 0.002
Accuracy Score: 0.92
True Positive Rate: 0.97
False Positive Rate: 0.18


The regularization coefficient: 0.028
Accuracy Score: 0.92
True Positive Rate: 0.91
False Positive Rate: 0.05
```




```
The best regularization coefficient: 0.199
Accuracy Score: 0.92
True Positive Rate: 0.91
False Positive Rate: 0.05
```

Problem 3: Gaussian Naive Bayes vs. Logistic Regression (10 pts)

Problem 3 (a) (5pts)

In this question, you will need to apply Gaussian Naïve Bayes classifier you implemented in Problem 1 on the `breast_cancer` dataset. There are two classes (i.e., $y = 0$ or $y = 1$). For each class, we make gaussian assumption on the likelihood, that is, $X|Y = c_k \sim \mathcal{N}(\mu_{c_k}, \Sigma)$ where $c_k = 0, 1$. Before performing a maximum a posteriori (MAP) estimation, we need to derive the posterior probability using the Gaussian Naïve Bayes assumption. Complete the implementation in the code block below.

Solution to Problem 3 Part (a)

```
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import load_breast_cancer

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

# Split the dataset
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

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

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


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

# For illustration, we can compare the predicted probabilities
print("---- label y = 0 ----")
probabilities_gnb0 = posterior_gnb[:, 0]


# And now, let's just print the first 10 predictions for comparison
print("Gaussian Naïve Bayes Probabilities (first 10):", probabilities_gnb0[:10])

print("---- label y = 1 ----")
probabilities_gnb1 = posterior_gnb[:, 1]


# And now, let's just print the first 10 predictions for comparison
print("Gaussian Naïve Bayes Probabilities (first 10):", probabilities_gnb1[:10])
```

```
---- label y = 0 ----
Gaussian Naïve Bayes Probabilities (first 10): [9.45898105e-10 1.00000000e+00 1.00000000e+00 7.13132757e-12
 7.61117487e-15 1.00000000e+00 1.00000000e+00 1.00000000e+00
 9.93714575e-01 1.88365303e-15]
---- label y = 1 ----
Gaussian Naïve Bayes Probabilities (first 10): [9.99999999e-001 3.43499533e-063 2.12542040e-012 1.00000000e+000
 1.00000000e+000 7.68406936e-123 3.57140845e-148 1.29771416e-021
 6.28542479e-003 1.00000000e+000]
```

Problem 3 (b) (5pts)

Now you will need to use logistic regression on the `breast_cancer` dataset for classification. Complete the code below. Compare the logistic regression probabilities with posterior probabilities based on Gaussian Naïve Bayes assumption in 3(a), are they similar or not? Reflect on why this is the case.

Solution to Problem 3 Part (b)

```
model = MyLogisticRegression(X_train, y_train)
model.fit(num_epochs = 50, step=2)
_, posterior_lr = model.predict(X_test)

probabilities_lr0 = 1-posterior_lr
probabilities_lr1 = posterior_lr

# For illustration, we can compare the predicted probabilities
print("---- label y = 0 ----")

print("Logistic Regression Probabilities (first 10):", probabilities_lr0[:10])

print("---- label y = 1 ----")

print("Logistic Regression Probabilities (first 10):", probabilities_lr1[:10])
```

```
---- label y = 0 ----
Logistic Regression Probabilities (first 10): [[7.88571065e-02]
 [9.99995242e-01]
 [9.99330470e-01]
 [4.79946170e-05]
 [7.63732795e-06]
 [1.00000000e+00]
 [9.99999999e-01]
 [9.53590633e-01]
 [2.91281917e-01]
 [1.82551167e-04]]
```

```
---- label y = 1 ----
Logistic Regression Probabilities (first 10): [[9.21142894e-01]
 [4.75783107e-06]
 [6.69530193e-04]
 [9.99952005e-01]
 [9.99992363e-01]
 [2.02404894e-12]
 [6.83409149e-10]
 [4.64093672e-02]
 [7.08718083e-01]
 [9.99817449e-01]]
```

The logistic regression probabilities are not similar to the posterior probabilities. The GNB assumes features which are independent and the likelihoods assume gaussian distribution.

Problem 4: Support Vector Machines (20 pts)

Support vector machine is an algorithm belonging to the group of the so called maximum margin classifiers. SVMs became popular in the early 90's due to their ability to solve classification, regression and novelty detection problems.

In Lecture 4, Page 29, you have seen that the SVM can be formulated as follows.

$$\underset{\mathbf{w},b}{\text{minimize}} \frac{1}{2}\|\mathbf{w}\|_2^2 + C\sum_{i=1}^{N}\max(0, 1 - y_i(\mathbf{w}^\top \mathbf{x}_i - b)) = \underset{\mathbf{w},b}{\text{minimize}}\, L_{hinge}(\mathbf{w})$$
$$\text{subject to } y_i\left(\mathbf{w}^\top \mathbf{x}_i - b\right) \geq 1 - \zeta_i, \quad \zeta_i \geq 0 \quad \forall i \in \{1, \ldots, n\}$$
$$\text{where } \zeta_i = \max(0, 1 - y_i(\mathbf{w}^\top \mathbf{x}_i - b))$$

In this formula, x_i's correspond to data samples. (For example, x_{23} corresponds to the 23rd data sample, a column vector with length D.) Similarly, y_i's are the ground truth target classes.

You will solve this optimization problem using the gradient descent algorithm in this example. The algorithm will work in batches, meaning that the parameters will not be updated after every sample, or the algorithm will not wait for all the training data to be used to calculate the loss. Instead, the parameter update will be performed after a predetermined number of training samples.

We need to calculate the gradient of the loss function L_{hinge} with respect to $\mathbf{w}$ and b. Considering the formulas presented in the matrix cookbook Page 10, the derivative of the max function, and the chain rule, we can find the partial derivatives as follows. First consider the derivative of the max function.

$$f_{max}(x) = \max(0, g(x))$$
$$f'_{max}(x) = \begin{cases} g'(x) & \text{if } g(x) \geq 0 \\ 0 & \text{if } g(x) \leq 0 \end{cases}$$

And the partial derivatives of the hinge loss is follows.

$$\frac{\partial L_{hinge}(\mathbf{w}, b)}{\partial \mathbf{w}} = \mathbf{w} + C\sum_{i=1}^{N} max'(0, -y_i\mathbf{x_i})$$
$$\frac{\partial L_{hinge}(\mathbf{w}, b)}{\partial b} = C\sum_{i=1}^{N} max'(0, y_i\mathbf{1})$$
$$\text{where } \mathbf{1} \text{ is a column vector with the same size of } b \text{ consisting of ones.}$$

Problem 4 (a) (18 Pts)

Carefully read the questions below and answer them appropriately.

i) Complete the `hingeloss` function.

ii) Calculate the code snippet where the gradients are calculated.

iii) Calculate the code snippet where weights and biases are updated.

iv) Complete the `predict` function.

> Do not change the function definitions for the functions defined in the SVM 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.

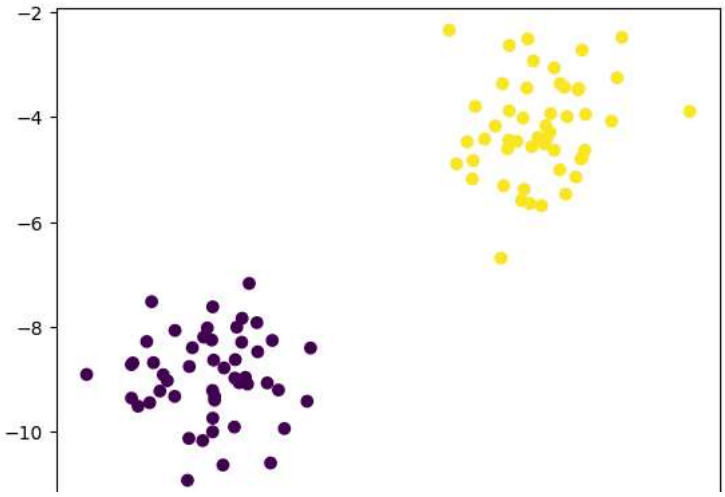
Solution to Problem 4 Part (a)

```
import numpy as np

class SVM:

    def __init__(self, C = 1.0):
        # C = error term
        self.C = C
        self.w = 0
        self.b = 0
    ## question i
    def hingeloss(self, w, b, x, y):
        """Function computes the hinge loss"""
        """
        Parameters:
        -----------
        w: ndarray of shape (D,)
            1D numpy array representing the normal vector to the SVM hyperplane.

        b: ndarray of shape (D,)
            1D numpy array representing the vector that translates the SVM hyperplane.

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

        Y: ndarray of shape (N,)
            1D numpy array containing containing ground truth class information.

        Returns:
        --------
        y_pred: float
            float containing the hinge loss.

        """
        # hinge loss function / calculation
        # regularization term
        hinge_loss = 0
        for i in range(x.shape[0]):
          margin = y[i] * (np.dot(w, x[i]) - b)
          hinge_loss += max(0, 1 - margin)
        regular_term = 0.5 * np.dot(w,w.T)
        loss = regular_term + self.C * hinge_loss
        return loss.item()


    def fit(self, X, Y, batch_size=100, learning_rate=0.001, epochs=1000):
        # the number of features in X
        number_of_features = X.shape[1]

        # the number of Samples in X
        number_of_samples = X.shape[0]

        c = self.C

        # creating ids from 0 to number_of_samples - 1
        ids = np.arange(number_of_samples)

        # shuffling the samples randomly
        np.random.shuffle(ids)

        # creating an array of zeros
        w = np.zeros((1, number_of_features))
        b = 0
        losses = []

        # gradient descent algorithm
        for i in range(epochs):
            # calculating the Hinge Loss
            l = self.hingeloss(w, b, X, Y)

            # appending all losses
            losses.append(l)
```

```
            # starting from 0 to the number of samples with batch_size as interval
            for batch_initial in range(0, number_of_samples, batch_size):
                gradw = np.zeros_like(w)
                gradb = 0

                for j in range(batch_initial, batch_initial+ batch_size):
                    if j < number_of_samples:
                        # The variable x corresponds to the index of the current
                        # data sample selected for the batch. X[x] corresponds
                        # to the data sample itself and Y[x] corresponds to the
                        # associated target class.
                        x = ids[j]
                        ## question ii
                        # calculating the gradients
                        margin = Y[x] * (np.dot(w, X[x]) - b)

                        # Notice that the summation of the gradients for a
                        # batch is already written. (gradw +=) What you need
                        # to do is to write the gradient for the current
                        # data sample.
                        if margin < 1:
                            gradw += -c * Y[x] * X[x]
                            gradb += -c * Y[x]
                gradw += w

                ## question iii
                # updating weights and bias
                w -= learning_rate * gradw
                b -= learning_rate * gradb

        self.w = w
        self.b = b

        return self.w, self.b, losses

    def predict(self, X):
        ## question iv
        # prediction
        """Function predicts the output of the SVM"""
        """
        Parameters:
        -----------
        X: ndarray of shape (N,D)
            2D numpy array containing N testing examples having D dimensions each.

        Returns:
        --------
        svm_prediction: ndarray of shape (N,)
            The prediction of the SVM algorithm in the form of integers. (The
            output should be an element of the set {1, -1}.)
        """
        decision_val = np.dot(X, self.w.T) + self.b
        svm_prediction = np.sign(decision_val)
        return svm_prediction

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

# Visualizing SVM
def visualize_svm():

    def get_hyperplane_value(x, w, b, offset):
        return (-w[0][0] * x + b + offset) / w[0][1]

    fig = plt.figure()
    ax = fig.add_subplot(1,1,1)
    plt.scatter(X_test[:, 0], X_test[:, 1], marker="o", c=y_test)

    x0_1 = np.amin(X_test[:, 0])
    x0_2 = np.amax(X_test[:, 0])

    x1_1 = get_hyperplane_value(x0_1, w, b, 0)
    x1_2 = get_hyperplane_value(x0_2, w, b, 0)

    x1_1_m = get_hyperplane_value(x0_1, w, b, -1)
    x1_2_m = get_hyperplane_value(x0_2, w, b, -1)

    x1_1_p = get_hyperplane_value(x0_1, w, b, 1)
```

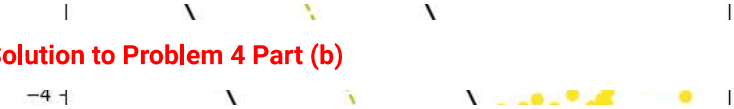
```
        x1_2_p = get_hyperplane_value(x0_2, w, b, 1)

        ax.plot([x0_1, x0_2], [x1_1, x1_2], "y--")
        ax.plot([x0_1, x0_2], [x1_1_m, x1_2_m], "k")
        ax.plot([x0_1, x0_2], [x1_1_p, x1_2_p], "k")

        x1_min = np.amin(X[:, 1])
        x1_max = np.amax(X[:, 1])
        ax.set_ylim([x1_min - 3, x1_max + 3])

        plt.show()

    from sklearn import datasets
    import matplotlib.pyplot as plt
    from sklearn.metrics import accuracy_score
    from sklearn.model_selection import train_test_split

    # Creating dataset
    X, y = datasets.make_blobs(

            n_samples = 100, # Number of samples
            n_features = 2, # Features
            centers = 2,
            cluster_std = 1,
            random_state=40
        )

    # Classes 1 and -1
    y = np.where(y == 0, -1, 1)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=42)

    svm = SVM()

    w, b, losses = svm.fit(X_train, y_train)

    prediction = svm.predict(X_test)

    # Loss value
    lss = losses.pop()

    print("Loss:", lss)
    # print("Prediction:", prediction)
    print("Accuracy:", accuracy_score(prediction, y_test))
    print("w, b:", [w, b])

    # Visualizing the scatter plot of the dataset
    def visualize_dataset():
        plt.scatter(X[:, 0], X[:, 1], c=y)


    visualize_dataset()
    visualize_svm()
```

```
Loss: 0.11540225591654017
Accuracy: 1.0
w, b: [array([[0.46008921, 0.13660546]]), 0.06300000000000004]
```



Problem 4 (b) (2 Pts)

In this part, you will use generate another dataset that is not completely separable and see the behavior of SVM. For this subproblem, adjust the number of data samples to 400 and cluster standard deviations as 4.



Solution to Problem 4 Part (b)


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

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


# Creating dataset
X, y = datasets.make_blobs(

        n_samples = 400,
        n_features = 2,
        centers = 2,
        cluster_std = 4,
        random_state=40
    )

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


# Classes 1 and -1
y = np.where(y == 0, -1, 1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.5, random_state=42)

svm = SVM()

w, b, losses = svm.fit(X_train, y_train)

prediction = svm.predict(X_test)

# Loss value
```