In [ ]:
```
# ALL IMPORTS
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import roc_curve, auc, log_loss, roc_auc_score, f1_scor
```

Assignment 3 - Supervised Learning: model training and evaluation

Jake Bova

Netid: jb240893

Names of students you worked with on this assignment: Bri, Brock

Note: this assignment falls under collaboration Mode 2: Individual Assignment – Collaboration Permitted. Please refer to the syllabus for additional information.

Instructions for all assignments can be found here, and is also linked to from the course syllabus.

Total points in the assignment add up to 90; an additional 10 points are allocated to presentation quality.

Learning Objectives:

This assignment will provide structured practice to help enable you to...

1. Understand the primary workflow in machine learning: (1) identifying a hypothesis function set of models, (2) determining a loss/cost/error/objective function to minimize, and (3) minimizing that function through gradient descent
2. Understand the inner workings of logistic regression and how linear models for classification can be developed.
3. Gain practice in implementing machine learning algorithms from the most basic building blocks to understand the math and programming behind them to achieve practical proficiency with the techniques
4. Implement batch gradient descent and become familiar with how that technique is used and its dependence on the choice of learning rate

5. Evaluate supervised learning algorithm performance through ROC curves and using cross validation
6. Apply regularization to linear models to improve model generalization performance

1

Classification using logistic regression: build it from the ground up

[60 points]

This exercise will walk you through the full life-cycle of a supervised machine learning classification problem. Classification problem consists of two features/predictors (e.g. petal width and petal length) and your goal is to predict one of two possible classes (class 0 or class 1). You will build, train, and evaluate the performance of a logistic regression classifier on the data provided. Before you begin any modeling, you'll load and explore your data in Part I to familiarize yourself with it - and check for any missing or erroneous data. Then, in Part II, we will review an appropriate hypothesis set of functions to fit to the data: in this case, logistic regression. In Part III, we will derive an appropriate cost function for the data (spoiler alert: it's cross-entropy) as well as the gradient descent update equation that will allow you to optimize that cost function to identify the parameters that minimize the cost for the training data. In Part IV, all the pieces come together and you will implement your logistic regression model class including methods for fitting the data using gradient descent. Using that model you'll test it out and plot learning curves to verify the model learns as you train it and to identify and appropriate learning rate hyperparameter. Lastly, in Part V you will apply the model you designed, implemented, and verified to your actual data and evaluate and visualize its generalization performance as compared to a KNN algorithm. **When complete, you will have accomplished learning objectives 1-5 above!**

I. Load, prepare, and plot your data

You are given some data for which you are tasked with constructing a classifier. The first step when facing any machine learning project: look at your data!

(a) Load the data.

- In the data folder in the same directory of this notebook, you'll find the data in `A3_Q1_data.csv`. This file contains the binary class labels, y, and the features x_1 and x_2.
- Divide your data into a training and testing set where the test set accounts for 30 percent of the data and the training set the remaining 70 percent.
- Plot the training data by class.

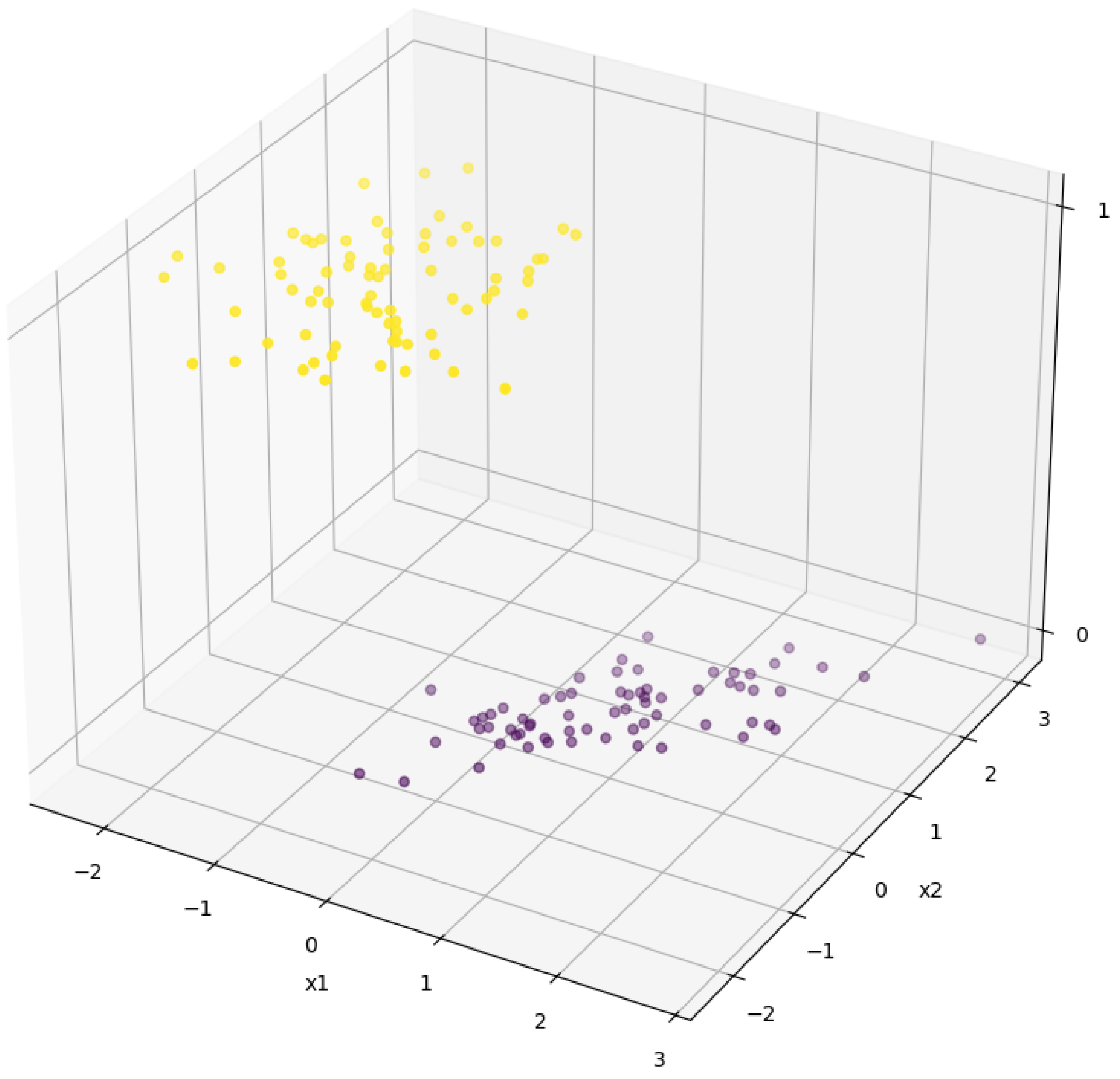
- Comment on the data: do the data appear separable? May logistic regression be a good choice for these data? Why or why not?

(b) Do the data require any preprocessing due to missing values, scale differences (e.g. different ranges of values), etc.? If so, how did you handle these issues?

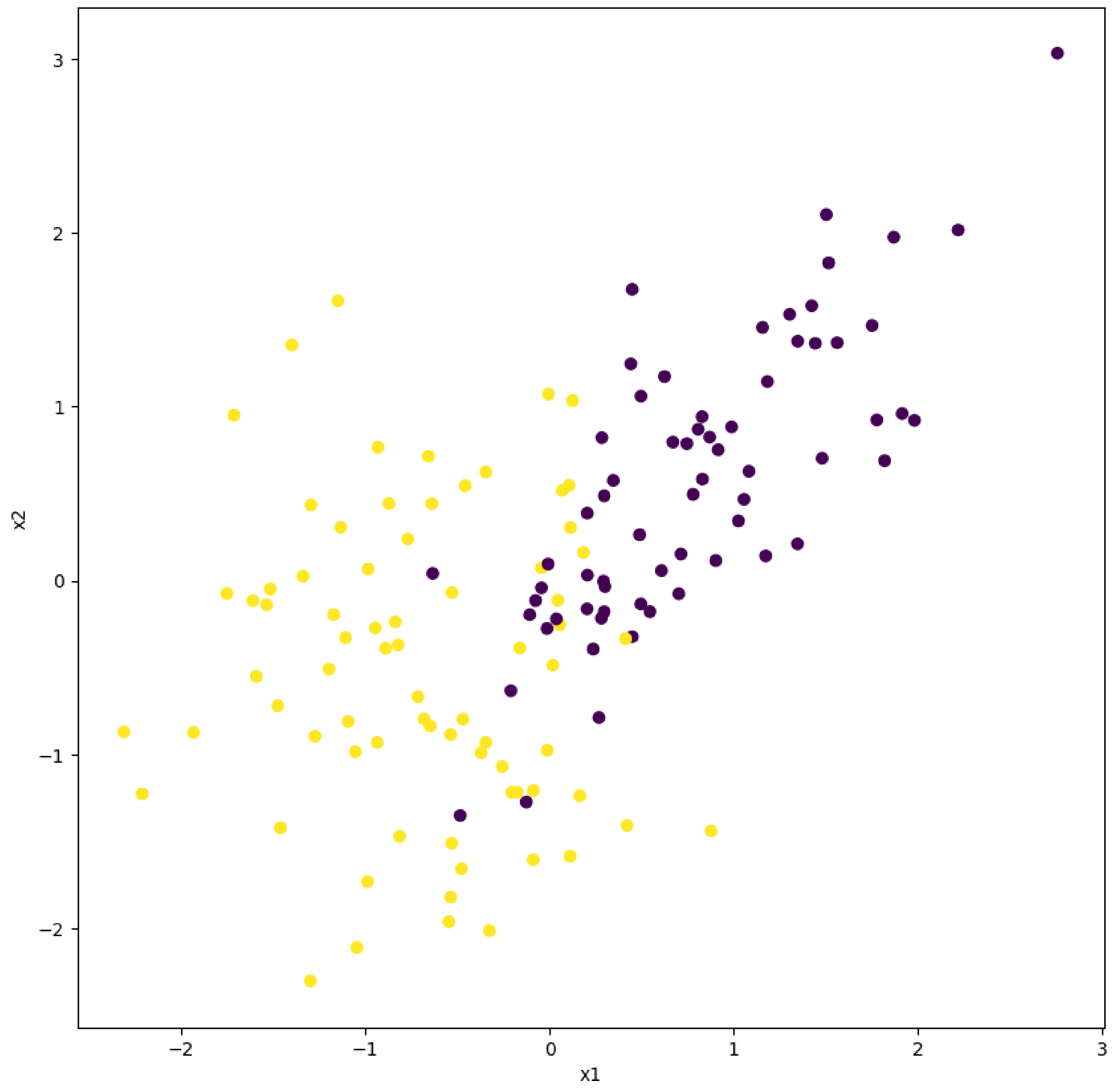
Next, we walk through our key steps for model fitting: choose a hypothesis set of models to train (in this case, logistic regression); identify a cost function to measure the model fit to our training data; optimize model parameters to minimize cost (in this case using gradient descent). Once we've completed model fitting, we will evaluate the performance of our model and compare performance to another approach (a KNN classifier).

START I ----------------

In [ ]:
```
# a) Load the data
data = pd.read_csv('data/A3_Q1_data.csv')
# a1) Split the data into training and test sets
# feature matrix X, target vector y
X = data.drop('y', axis=1)
y = data['y']
# split data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, rar
# normalize data
X_train = (X_train - X_train.mean()) / X_train.std()
X_test = (X_test - X_test.mean()) / X_test.std()
# a2) plot the training data by class
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X_train['x1'], X_train['x2'], y_train, c=y_train, cmap='viridis')
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('y')
ax.set_zticks([0, 1])
plt.show()
```



```
In [ ]: # graph the data in 2d colored by class (0 purple, 1 yellow)
        fig = plt.figure(figsize=(10, 10))
        ax = fig.add_subplot(111)
        ax.scatter(X_train['x1'], X_train['x2'], c=y_train, cmap='viridis')
        ax.set_xlabel('x1')
        ax.set_ylabel('x2')
        plt.show()
```



```
In [ ]: # b) assess data
        print(data.isna().sum()) # no missing values
```

END I ----------------

II. Stating the hypothesis set of models to evaluate (we'll use logistic regression)

Given that our data consists of two features, our logistic regression problem will be applied to a two-dimensional feature space. Recall that our logistic regression model is:

$$f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{w}^\top \mathbf{x}_i)$$

where the sigmoid function is defined as $\sigma(x) = \dfrac{e^x}{1+e^x} = \dfrac{1}{1+e^{-x}}$. Also, since this is a two-dimensional problem, we define $\mathbf{w}^\top \mathbf{x}_i = w_0 x_{i,0} + w_1 x_{i,1} + w_2 x_{i,2}$ and here, $\mathbf{x}_i = [x_{i,0}, x_{i,1}, x_{i,2}]^\top$, and $x_{i,0} \triangleq 1$

Remember from class that we interpret our logistic regression classifier output (or confidence score) as the conditional probability that the target variable for a given sample y_i is from class "1", given the observed features, $\mathbf{x}_i$. For one sample, $(y_i, \mathbf{x}_i)$, this is given as:

$$P(Y = 1|X = \mathbf{x}_i) = f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{w}^\top \mathbf{x}_i)$$

In the context of maximizing the likelihood of our parameters given the data, we define this to be the likelihood function $L(\mathbf{w}|y_i, \mathbf{x}_i)$, corresponding to one sample observation from the training dataset.

Aside: the careful reader will recognize this expression looks different from when we talk about the likelihood of our data given the true class label, typically expressed as $P(x|y)$, or the posterior probability of a class label given our data, typically expressed as $P(y|x)$. In the context of training a logistic regression model, the likelihood we are interested in is the likelihood function of our logistic regression ***parameters***, $\mathbf{w}$. *It's our goal to use this to choose the parameters to maximize the likelihood function.*

No output is required for this section - just read and use this information in the later sections.

III. Find the cost function that we can use to choose the model parameters, $\mathbf{w}$, that best fit the training data.

(c) What is the likelihood function that corresponds to all the N samples in our training dataset that we will wish to maximize? Unlike the likelihood function written above which gives the likelihood function for a *single training data pair* $(y_i, \mathbf{x}_i)$, this question asks for the likelihood function for the *entire training dataset* $\{(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \dots, (y_N, \mathbf{x}_N)\}$.

(d) Since a logarithm is a monotonic function, maximizing the $f(x)$ is equivalent to maximizing $\ln[f(x)]$. Express the likelihood from the last question as a cost function of the model parameters, $C(\mathbf{w})$; that is the negative of the logarithm of the likelihood. Express this cost as an average cost per sample (i.e. divide your final value by N), and use this quantity going forward as the cost function to optimize.

(e) Calculate the gradient of the cost function with respect to the model parameters $\nabla_{\mathbf{w}} C(\mathbf{w})$. Express this in terms of the partial derivatives of the cost function with respect to each of the parameters, e.g. $\nabla_{\mathbf{w}} C(\mathbf{w}) = \left[\frac{\partial C}{\partial w_0}, \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}\right]$.

To simplify notation, please use $\mathbf{w}^\top \mathbf{x}$ instead of writing out $w_0 x_{i,0} + w_1 x_{i,1} + w_2 x_{i,2}$ when it appears each time (where $x_{i,0} = 1$ for all i). You are also welcome to use $\sigma()$ to represent the sigmoid function. Lastly, this will be a function the features, $x_{i,j}$ (with the first index in the subscript representing the observation and the second the feature; targets, y_i; and the logistic regression model parameters, w_j.

(f) Write out the gradient descent update equation. This should clearly express how to update each weight from one step in gradient descent $w_j^{(k)}$ to the next $w_j^{(k+1)}$. There should be one equation for each model logistic regression model parameter (or you can represent it in vectorized form). Assume that η represents the learning rate.

START III ------------------------------

c) The likelihood function for all N samples can be expressed as the product of the individual likelihood functions for each data point (under the assumtion that the data points are independent and identically distributed).

Given sigmoid function:

$$f(\mathbf{x}_i, \mathbf{w}) = \sigma(\mathbf{w}^\top \mathbf{x}_i) = \frac{1}{1 + e^{-\mathbf{w}^\top \mathbf{x}_i}}$$

The likelihood for a single data point $(y_i, \mathbf{x}_i)$ is given by:

$$L(\mathbf{w}|y_i, \mathbf{x}_i) = \sigma(\mathbf{w}^\top \mathbf{x}_i)^{y_i}(1 - \sigma(\mathbf{w}^\top \mathbf{x}_i))^{1-y_i}$$

This equation reflects the binary nature of the data, where y_i is either 0 or 1, so the equation is similar to the Bernoulli distribution.

Since the data points are independent and identically distributed, we can multiply the likelihoods for each data point to get the likelihood for the entire dataset:

$$L(\mathbf{w}|y_i, \mathbf{x}_i) = \prod_{i=1}^{N} \sigma(\mathbf{w}^\top \mathbf{x}_i)^{y_i}(1 - \sigma(\mathbf{w}^\top \mathbf{x}_i))^{1-y_i}$$

d) In order to express the likelihood function as a cost function (using log likelihood), we take the negative log of the likelihood function (on the parameter $\mathbf{w}$):

$$C(\mathbf{w}) = -\ln L(\mathbf{w}|y_i, \mathbf{x}_i) = -\sum_{i=1}^{N} y_i \ln \sigma(\mathbf{w}^\top \mathbf{x}_i) + (1 - y_i)\ln(1 - \sigma(\mathbf{w}^\top \mathbf{x}_i))$$

Note that taking the logarithm of this function allows us to move the exponent from the sigmoid function to the outside of the function, which makes it easier to take the derivative.

From this, we divide by N to get the average cost per sample:

- (I am defining $\hat{y} = \sigma(\mathbf{w}^\top \mathbf{x}_i)$ where applicable)

$$C(\mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N} y_i \ln \sigma(\mathbf{w}^\top \mathbf{x}_i) + (1 - y_i)\ln(1 - \sigma(\mathbf{w}^\top \mathbf{x}_i))$$

e) To calculate the gradient of the cost function with respect to the model parameters $\nabla_{\mathbf{w}} C(\mathbf{w})$, we express it in terms of the partial derivatives of the cost function with respect to each of the parameters, e.g. $\nabla_{\mathbf{w}} C(\mathbf{w}) = \left[\frac{\partial C}{\partial w_0}, \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}\right]$:

- 1. setup:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{N}\sum_{i=1}^{N} y_i \frac{\partial}{\partial w_j} \ln \sigma(\mathbf{w}^\top \mathbf{x}_i) + (1 - y_i)\frac{\partial}{\partial w_j} \ln(1 - \sigma(\mathbf{w}^\top \mathbf{x}_i))$$

- 1.1) take the derivatives of the log-sigmoid function (a) and log of 1-sigmoid function (b):

- (a)

$$\frac{\partial}{\partial w_j} \ln \sigma(\mathbf{w}^\top \mathbf{x}_i) = \frac{\mathbf{x}_{ij}}{\sigma(\mathbf{w}^\top \mathbf{x}_i)}$$

- (b)

$$\frac{\partial}{\partial w_j} \ln(1 - \sigma(\mathbf{w}^\top \mathbf{x}_i)) = -\frac{\mathbf{x}_{ij}}{1 - \sigma(\mathbf{w}^\top \mathbf{x}_i)}$$

- 2. substitute (a) and (b) into the setup:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{N}\sum_{i=1}^{N} \left(y_i \frac{\mathbf{x}_{ij}}{\sigma(\mathbf{w}^\top \mathbf{x}_i)} - (1 - y_i)\frac{\mathbf{x}_{ij}}{1 - \sigma(\mathbf{w}^\top \mathbf{x}_i)} \right)$$

- 3. combine terms:

$$\frac{\partial C}{\partial w_j} = -\frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_{ij} \left(y_i \frac{1}{\sigma(\mathbf{w}^\top \mathbf{x}_i)} - (1 - y_i)\frac{1}{1 - \sigma(\mathbf{w}^\top \mathbf{x}_i)} \right)$$

- 4. we can simplify the expression in parentheses (by performing the arithmetic then cancelling terms):

$$\frac{\partial C}{\partial w_j} = -\frac{1}{N}\sum_{i=1}^{N} \mathbf{x}_{ij} \left(y_i - \sigma(\mathbf{w}^\top \mathbf{x}_i) \right)$$

- 5. the gradient vector would be:

$$\nabla_{\mathbf{w}} C(\mathbf{w}) = \frac{1}{N}\sum_{i=1}^{N} (\sigma(\mathbf{w}^\top \mathbf{x}_i) - y_i)\mathbf{x}_{ij}$$

$$for\ x = \left[\frac{\partial C}{\partial w_0}, \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}\right]$$

Where:

- $\mathbf{x}_{ij}$ is the j^{th} feature of the i^{th} data point
- $\mathbf{w}^\top \mathbf{x}_i$ is the dot product of the weight vector and the feature vector for the i^{th} data point
- $\sigma(\mathbf{w}^\top \mathbf{x}_i)$ is the sigmoid function applied to the dot product of the weight vector and the feature vector for the i^{th} data point
- y_i is the target value for the i^{th} data point

f) The gradient descent update equation is used to iteratively adjust model params (weights) to minimize the cost function. The update equation is:

$$w_j^{(k+1)} = w_j^{(k)} - \eta \nabla_{\mathbf{w}} C(\mathbf{w})$$

Where:

- η is the learning rate.
- $w_j^{(k)}$ is the value of the weight w_j at iteration k.
- $w_j^{(k+1)}$ is the value of the updated weight w_j at iteration $k+1$.
- $\frac{\partial C}{\partial w_j}$ is the partial derivative of the cost function with respect to the weight w_j (see above)

END III --------------------------------

IV. Implement gradient descent and your logistic regression algorithm

(g) Implement your logistic regression model.

- You are provided with a template, below, for a class with key methods to help with your model development. It is modeled on the Scikit-Learn convention. For this, you only need to create a version of logistic regression for the case of two feature variables (i.e. two predictors).
- Create a method called `sigmoid` that calculates the sigmoid function
- Create a method called `cost` that computes the cost function $C(\mathbf{w})$ for a given dataset and corresponding class labels. This should be the **average cost** (make sure your total cost is divided by your number of samples in the dataset).
- Create a method called `gradient_descent` to run **one step** of gradient descent on your training data. We'll refer to this as "batch" gradient descent since it takes into account the gradient based on all our data at each iteration of the algorithm.
- Create a method called `fit` that fits the model to the data (i.e. sets the model parameters to minimize cost) using your `gradient_descent` method. In doing this we'll need to make some assumptions about the following:

    - Weight initialization. What should you initialize the model parameters to? For this, randomly initialize the weights to a different values between 0 and 1.
    - Learning rate. How slow/fast should the algorithm step towards the minimum? This you will vary in a later part of this problem.
    - Stopping criteria. When should the algorithm be finished searching for the optimum? There are two stopping criteria: small changes in the gradient descent step size and a maximum number of iterations. The first is whether there was a sufficiently small change in the gradient; this is evaluated as whether the magnitude of the step that the gradient descent algorithm takes changes by less than 10^{-6} between iterations. Since we have a weight vector, we can compute the change in the weight by evaluating the L_2 norm (Euclidean norm) of the change in the vector between iterations. From our gradient descent update equation we know that mathematically this is $|| - \eta \nabla_{\mathbf{w}} C(\mathbf{w})||$. The second criterion is met if a maximum number of iterations has been reach (5,000 in this case, to prevent infinite loops from poor choices of learning rates).
    - Design your approach so that at each step in the gradient descent algorithm you evaluate the cost function for both the training and the test data for each new value for the model weights. You should be able to plot cost vs gradient descent iteration for both the training and the test data. This will allow you to plot "learning curves" that can be informative for how the model training process is proceeding.
- Create a method called `predict_proba` that predicts confidence scores (that can be thresholded into the predictions of the `predict` method.
- Create a method called `predict` that makes predictions based on the trained model, selecting the most probable class, given the data, as the prediction, that is class that yields the larger $P(y|\mathbf{x})$.
- (Optional, but recommended) Create a method called `learning_curve` that produces the cost function values that correspond to each step from a previously run gradient descent operation.
- (Optional, but recommended) Create a method called `prepare_x` which appends a column of ones as the first feature of the dataset $\mathbf{X}$ to account for the bias term ($x_{i,1} = 1$).

This structure is strongly encouraged; however, you're welcome to adjust this to your needs (adding helper methods, modifying parameters, etc.).

Class: LogisticRegression

```
Method Initialize():
    weights <- None
    savedWeights <- Empty List
    savedTrainCosts <- Empty List
    savedTestCosts <- Empty List
```

```
Method Sigmoid(X, weights):
    Return 1 / (1 + e^(-X * weights))

Method Cost(X, y, weights):
    Return -Mean(y * log(Sigmoid(X, weights)) + (1-y) * log(1
- Sigmoid(X, weights)))

Method GradientDescent(X, y, learningRate):
    numSamples <- Number of rows in X

    gradient <- Transpose(X) * (Sigmoid(X, weights) - y) /
numSamples

    oldWeights <- Copy of weights

    weights <- weights - learningRate * gradient

    Append weights to savedWeights

    Return EuclideanNorm of (weights - oldWeights)

Method Fit(X, y, initialWeights, learningRate,
deltaThreshold=1e-6, maxIterations=5000, verbose=False):
    X <- AddColumnOfOnes to X

    weights <- Randomly initialize weights based on number of
features in X

    weightChange <- Infinity

    iterationCount <- 0

    WHILE weightChange > deltaThreshold AND iterationCount <
maxIterations:
        weightChange <- GradientDescent(X, y, learningRate)

        trainCost <- Cost(X, y, weights)

        Append trainCost to savedTrainCosts

        Increment iterationCount

        IF verbose:
            Print weights

Method PredictProba(X):
    X <- AddColumnOfOnes to X

    Return Sigmoid(X, weights)

Method Predict(X, threshold=0.5):
    confidenceScore <- PredictProba(X)
```

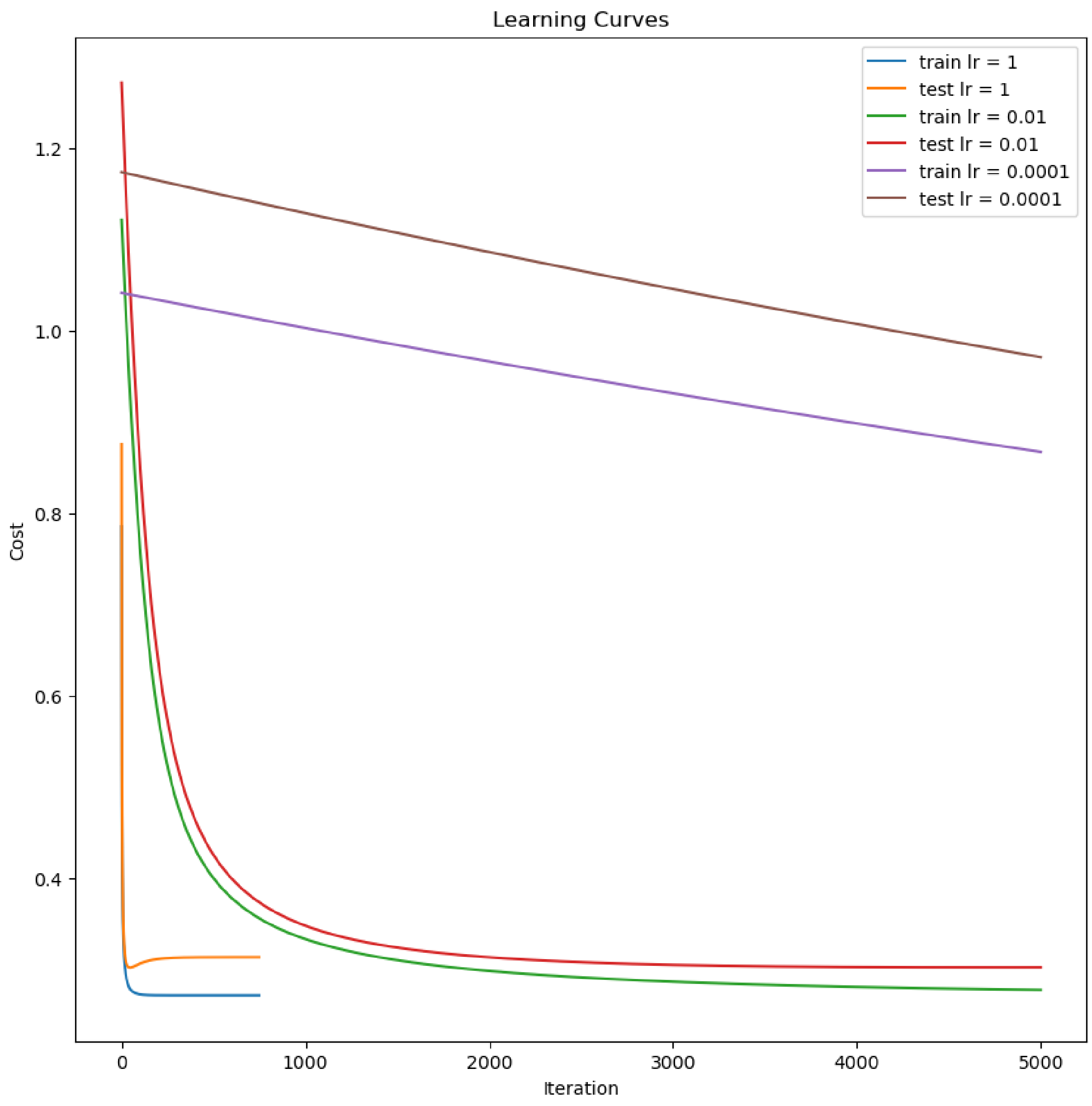
```
        Return 1 if confidenceScore >= threshold else 0 for each
    score in confidenceScore

    Method LearningCurve(X, y):
        X <- AddColumnOfOnes to X

        trainCosts <- Empty List

        FOR each weight in savedWeights:
            Append Cost(X, y, weight) to trainCosts

        Return trainCosts

    Method AddColumnOfOnes(X):
        bias <- Create column of ones with number of rows equal
    to X

        Return Concatenate bias and X column-wise
```

In [ ]:
```python
# Logistic regression class
class Logistic_regression:
    # Class constructor
    def __init__(self):
        self.w = None     # logistic regression weights
        self.saved_w = [] # Since this is a small problem, we can save the w
                          #  at each iteration of gradient descent to build
                          #  learning curves
        self.saved_train_costs = []
        self.saved_test_costs = []
        # returns nothing
        pass

    # Method for calculating the sigmoid function of w^T X for an input set
    def sigmoid(self, X, w):
        # returns the value of the sigmoid
        sigmoid = 1 / (1 + np.exp(-np.dot(X, w)))
        return sigmoid

    # Cost function for an input set of weights
    def cost(self, X, y, w):
        # returns the average cross entropy cost
        avg_cost = -np.mean(y * np.log(self.sigmoid(X, w)) + (1 - y) * np.lo
        return avg_cost

    # Update the weights in an iteration of gradient descent
    def gradient_descent(self, X, y, lr):
        # returns a scalar of the magnitude of the Euclidean norm
        #  of the change in the weights during one gradient descent step
        m = X.shape[0] # number of samples
        # calculate the gradient
        grad = np.dot(X.T, (self.sigmoid(X, self.w) - y)) / m
        # save old weights
        old_w = self.w.copy()
        # update the weights
        self.w = self.w - (lr * grad)
        # save the weights
        self.saved_w.append(self.w)
        # calculate the magnitude of the change in the weights
        weight_change = np.linalg.norm(self.w - old_w)
        # weight_change = np.linalg.norm(lr * grad)
        return weight_change

    # Fit the logistic regression model to the data through gradient descent
    def fit(self, X, y, w_init, lr, delta_thresh=1e-6, max_iter=5000, verbos
        # Note the verbose flag enables you to print out the weights at each
        #  (optional - but may help with one of the questions)
        X = self.prepare_x(X) # add a column of ones to X

        self.w = np.random.rand(X.shape[1]) # initialize the weights

        # initialize the weight change
        weight_change = np.inf
        # initialize the iteration count
        iter_count = 0
        # while the weight change is greater than the threshold and the iter
        while weight_change > delta_thresh and iter_count < max_iter:
            # update the weights
            weight_change = self.gradient_descent(X, y, lr)
            # calculate the cost
            train_cost = self.cost(X, y, self.w)
            # save the costs
            self.saved_train_costs.append(train_cost)
            # increment the iteration count
            iter_count += 1
            # print the weights if verbose
            if verbose:
                print(self.w)
        # returns nothing
        pass

    # Use the trained model to predict the confidence scores (prob of positi
    def predict_proba(self, X):
        # returns the confidence score for the each sample
        X = self.prepare_x(X) # add a column of ones to X
        confidence_score = self.sigmoid(X, self.w)
        return confidence_score

    # Use the trained model to make binary predictions
    def predict(self, X, thresh=0.5):
        # returns a binary prediction for each sample
        confidence_score = self.predict_proba(X) # get the confidence score
        binary_pred = np.where(confidence_score >= thresh, 1, 0) # make the
        return binary_pred

    # Stores the learning curves from saved weights from gradient descent
    def learning_curve(self, X, y):
        # returns the value of the cost function from each step in gradient
        #  from the last model fitting process
        X = self.prepare_x(X)
        train_costs = []
        for w in self.saved_w:
            train_costs.append(self.cost(X, y, w)) # calculate the cost for
        return train_costs

    # Appends a column of ones as the first feature to account for the bias
    def prepare_x(self, X):
        # returns the X with a new feature of all ones (a column that is the
        bias = np.ones((X.shape[0], 1)) # ones creates an array of ones in t
        return np.hstack([bias, X]) # hstack stacks arrays in sequence horiz
```

(h) Choose a learning rate and fit your model. Learning curves are a plot of metrics of model performance evaluated through the process of model training to provide insight about how model training is proceeding. Show the learning curves for the gradient descent process for learning rates of $\{10^{-0}, 10^{-2}, 10^{-4}\}$. For each learning rate plot the learning curves by plotting **both the training and test data average cost** as a function of each iteration of gradient descent. You should run the model fitting process until it completes (up to 5,000 iterations of gradient descent). Each of the 6 resulting curves (train and test average cost for each learning rate) should be plotted on the **same set of axes** to enable direct comparison. *Note: make sure you're using average cost per sample, not the total cost.*

- Try running this process for a really big learning rate for this problem: 10^2. Look at the weights that the fitting process generates over the first 50 iterations and how they change. Either print these first 50 iterations as console output or plot them. What happens? How does the output compare to that corresponding to a learning rate of 10^0 and why?
- What is the impact that the different values of learning have on the speed of the process and the results?
- Of the options explored, what learning rate do you prefer and why?
- Use your chosen learning rate for the remainder of this problem.

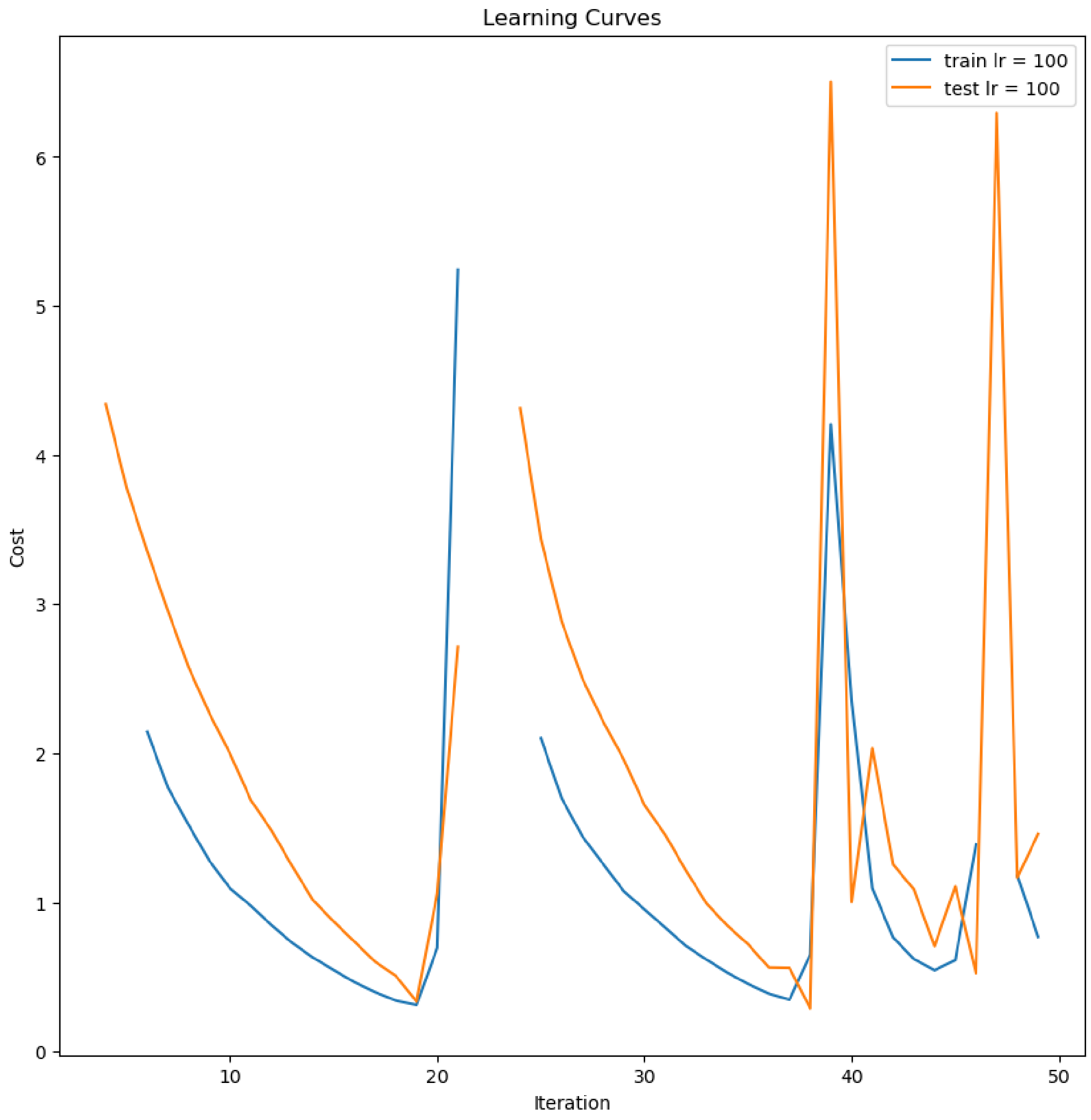
In [ ]:
```
# h) Plot the learning curves
# plot the learning curves for learning rates of 1, 0.01, and 0.0001
fig = plt.figure(figsize=(10, 10))
for lr in [1, 0.01, 0.0001]:
    logreg = Logistic_regression()
    logreg.fit(X_train, y_train, w_init=0, lr=lr, max_iter=5000, verbose=Fal
    plt.plot(logreg.learning_curve(X_train, y_train), label='train lr = ' +
    plt.plot(logreg.learning_curve(X_test, y_test), label='test lr = ' + str
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.title('Learning Curves')
plt.legend()
plt.show()
```



For the learning rates shown above, I would select the 0.01 rate. Lower rates take too long to converge, and may not provide the best results. Higher rates converge too quickly, and may prevent the model from finding the best solution.

In [ ]:
```
# lr of 100
logreg4 = Logistic_regression()
logreg4.fit(X_train, y_train, w_init=0, lr=100, max_iter=5000, verbose=False
logreg4.predict(X_test)
# accuracy
print('Accuracy: ', np.mean(logreg4.predict(X_test) == y_test))
# get learning curve
lclr4train = logreg4.learning_curve(X_train, y_train)
lclr4test = logreg4.learning_curve(X_test, y_test)

# plot the first 50 iterations
fig = plt.figure(figsize=(10, 10))
plt.plot(lclr4train[:50], label='train lr = 100')
plt.plot(lclr4test[:50], label='test lr = 100')
plt.xlabel('Iteration')
plt.ylabel('Cost')
plt.title('Learning Curves')
plt.legend()
plt.show()
```

```
/tmp/ipykernel_49010/1254693442.py:23: RuntimeWarning: divide by zero encoun
tered in log
  avg_cost = -np.mean(y * np.log(self.sigmoid(X, w)) + (1 - y) * np.log(1 -
self.sigmoid(X, w)))
Accuracy:  0.8333333333333334
```



I'm not exactly sure what is happening here. It seems that the learning rate of 100 is too high, and the model is overshooting the minimum. This is causing the cost to increase, and the weights to oscillate. The learning rate of 1 is also too high, but not as bad as 100. The learning rate of 0.01 is much better, and the cost is decreasing as expected.

V. Evaluate your model performance through cross validation

(i) Test the performance of your trained classifier using K-folds cross validation resampling technique. The scikit-learn package StratifiedKFolds may be helpful.

- Train your logistic regression model and a K-Nearest Neighbor classification model with $k = 7$ nearest neighbors.
- Using the trained models, make four plots: two for logistic regression and two for KNN. For each model have one plot showing the training data used for fitting the model, and the other showing the test data. On each plot, include the decision boundary resulting from your trained classifier.
- Produce a Receiver Operating Characteristic curve (ROC curve) that represents the performance from cross validated performance evaluation for each classifier (your logistic regression model and the KNN model, with $k = 7$ nearest neighbors). For the cross validation, use $k = 10$ folds.
  - Plot these curves on the same set of axes to compare them
  - On the ROC curve plot, also include the chance diagonal for reference (this represents the performance of the worst possible classifier). This is represented as a line from $(0, 0)$ to $(1, 1)$.
  - Calculate the Area Under the Curve for each model and include this measure in the legend of the ROC plot.
- Comment on the following:
  - What is the purpose of using cross validation for this problem?
  - How do the models compare in terms of performance (both ROC curves and decision boundaries) and which model (logistic regression or KNN) would you select to use on previously unseen data for this problem and why?

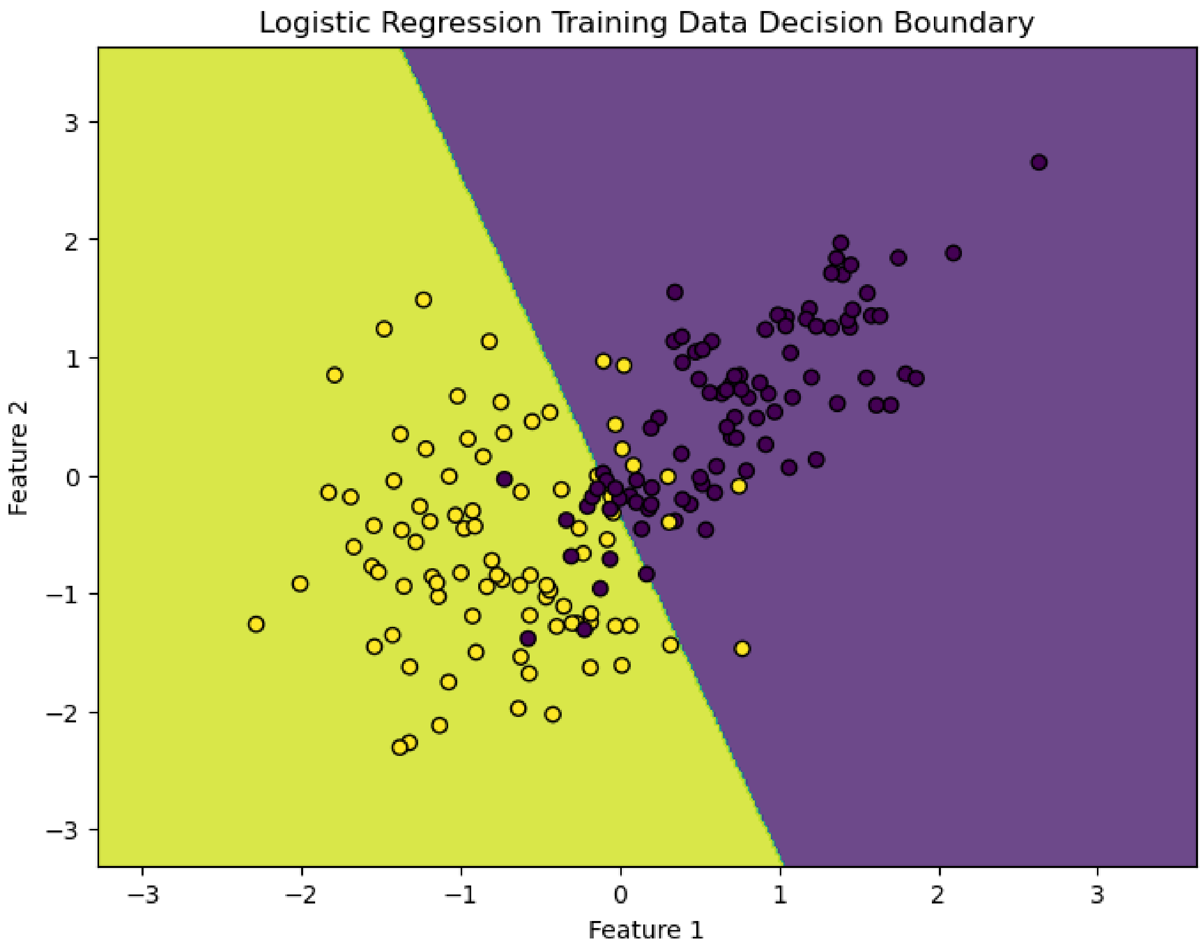
ANSWER

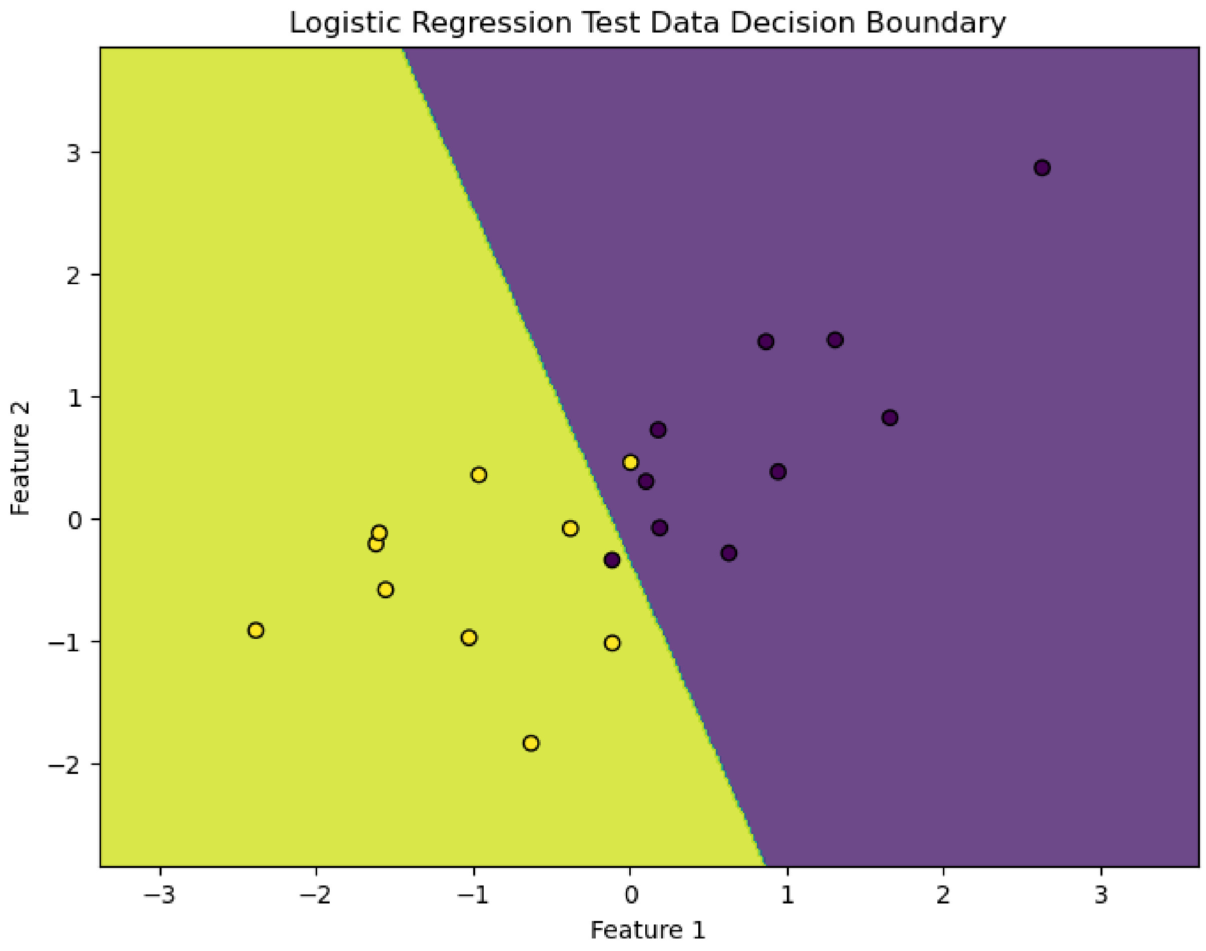
In [ ]:
```python
X = (X - X.mean()) / X.std() # normalize the data

X_np = X.to_numpy() # convert to numpy arrays
y_np = y.to_numpy()

# Initialize the models
log_reg = Logistic_regression()
knn = KNeighborsClassifier(n_neighbors=7)

# Initialize StratifiedKFolds
skf = StratifiedKFold(n_splits=10)

log_reg_tprs = [] # true positive rates
log_reg_aucs = [] # area under the curve
knn_tprs = [] # true positive rates
knn_aucs = [] # area under the curve
mean_fpr = np.linspace(0, 1, 100) # mean false positive rate
```

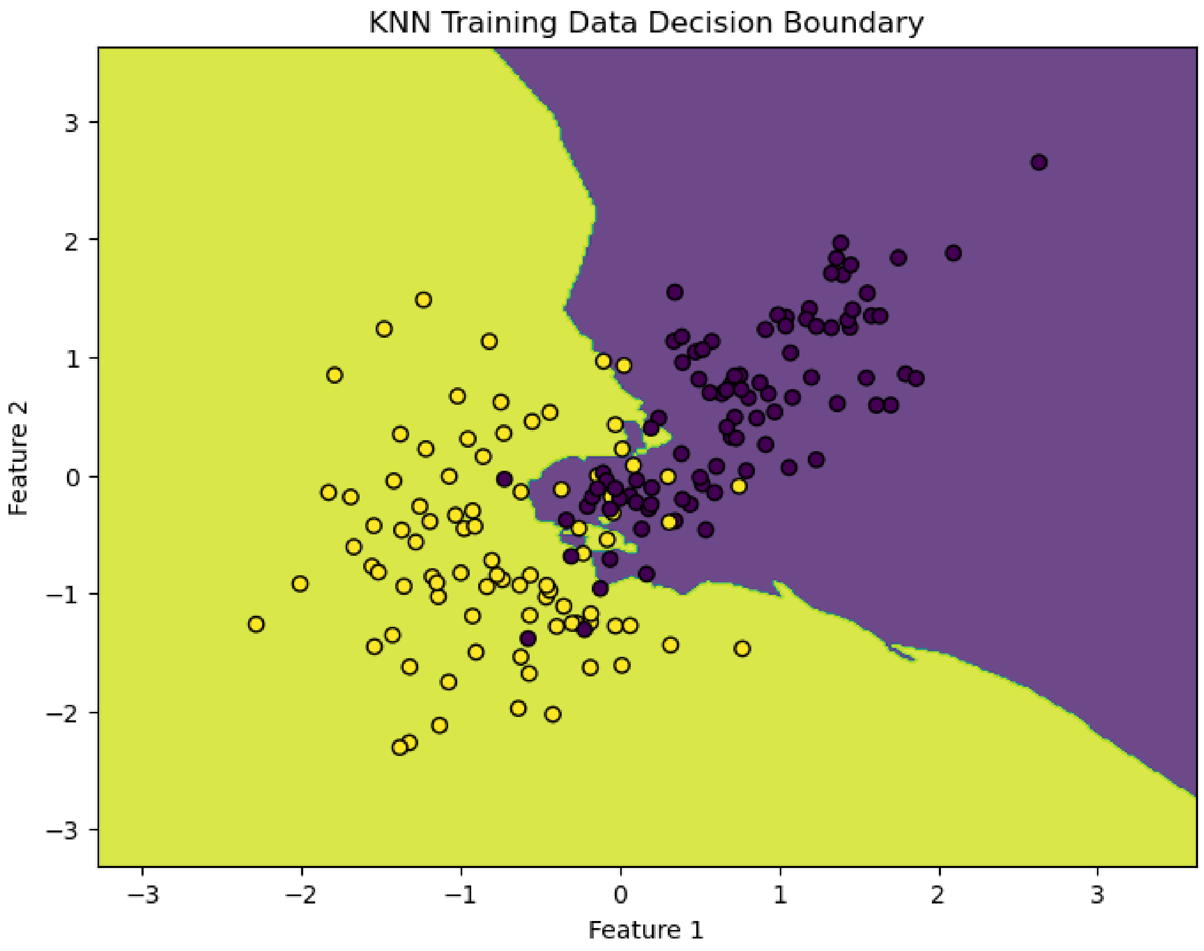
```
for train_index, test_index in skf.split(X_np, y_np):
    X_train, X_test = X_np[train_index], X_np[test_index]
    y_train, y_test = y_np[train_index], y_np[test_index]

    # Train logistic regression
    log_reg.fit(X_train, y_train, w_init=None, lr=0.01)
    y_pred_prob_log = log_reg.predict_proba(X_test)
    fpr, tpr, _ = roc_curve(y_test, y_pred_prob_log)
    log_reg_tprs.append(np.interp(mean_fpr, fpr, tpr)) # interpolate the tru
    log_reg_aucs.append(auc(fpr, tpr)) # calculate the area under the curve

    # Train KNN
    knn.fit(X_train, y_train)
    y_pred_prob_knn = knn.predict_proba(X_test)[:, 1]
    fpr, tpr, _ = roc_curve(y_test, y_pred_prob_knn)
    knn_tprs.append(np.interp(mean_fpr, fpr, tpr))
    knn_aucs.append(auc(fpr, tpr))
```

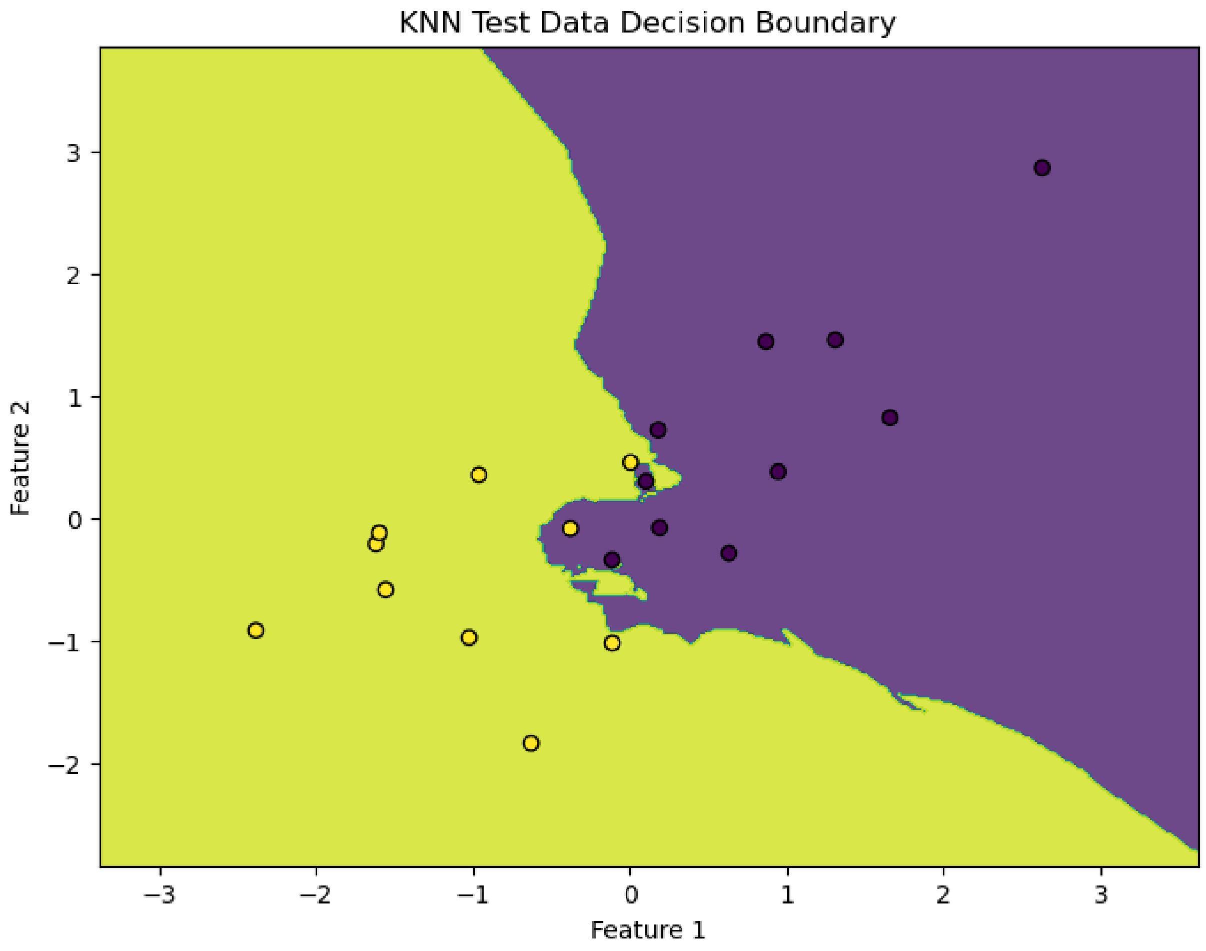
In [ ]:

```
# CODE FOR PLOTTING DECISION BOUNDARIES IS SOURCED FROM MY DATA SCIENCE ANAL

def plot_decision_boundary(model, X, y, model_name="Model"):
    plt.figure(figsize=(8, 6))
    h = .02
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

    if model_name == "Logistic Regression":
        Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    else:
        Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)

    plt.contourf(xx, yy, Z, alpha=0.8)
    plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors='k', marker='o')
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title(model_name + ' Decision Boundary')
    plt.show()

# Plot decision boundaries for logistic regression and KNN
plot_decision_boundary(log_reg, X_train, y_train, "Logistic Regression Trair
plot_decision_boundary(log_reg, X_test, y_test, "Logistic Regression Test Da
plot_decision_boundary(knn, X_train, y_train, "KNN Training Data")
plot_decision_boundary(knn, X_test, y_test, "KNN Test Data")
```

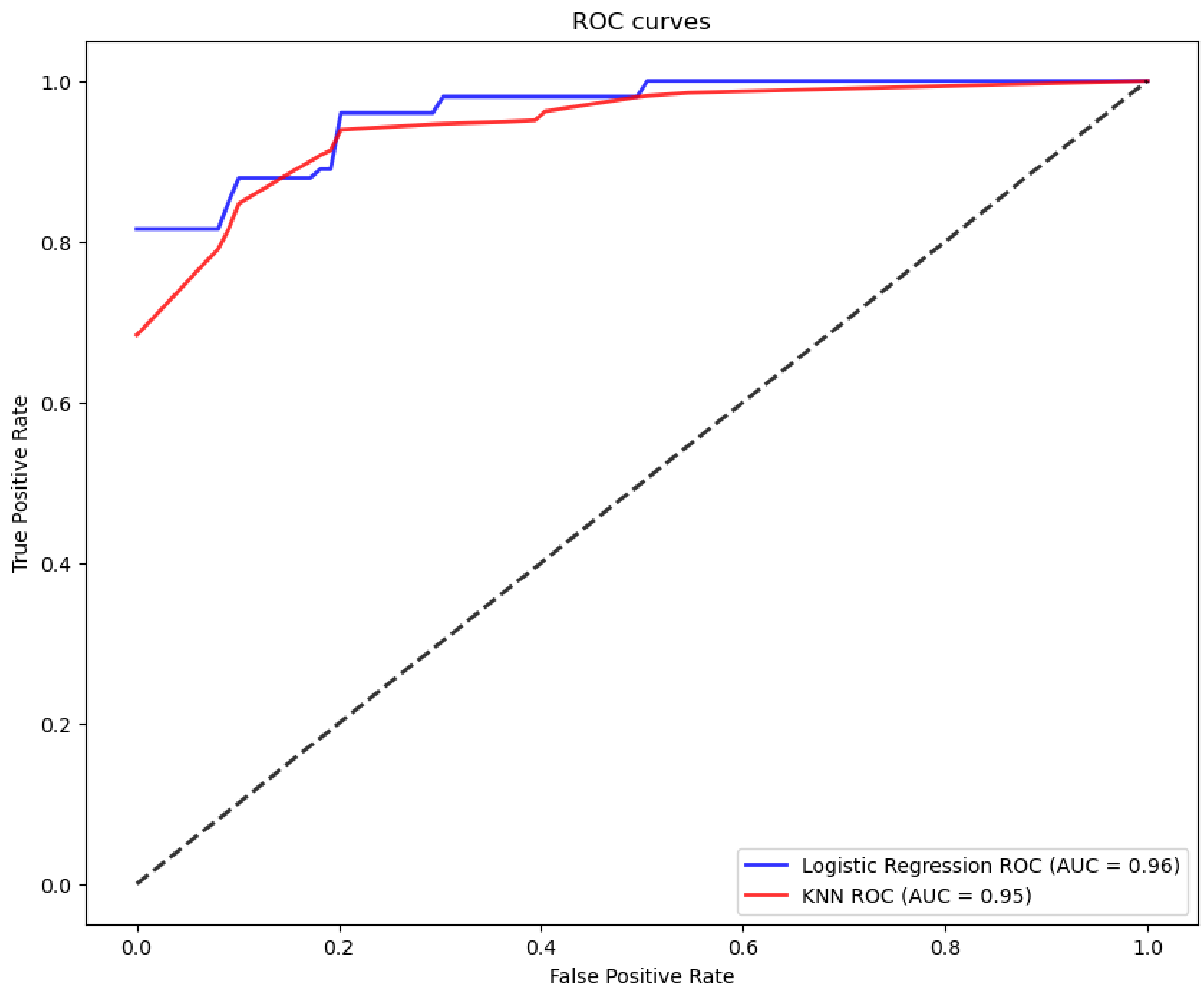

Logistic Regression Training Data Decision Boundary
Feature 2
Feature 1
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3


Logistic Regression Test Data Decision Boundary
Feature 1
Feature 2
−3
−2
−1
0
1
2
3


KNN Training Data Decision Boundary
Feature 1
Feature 2
−3
−2
−1
0
1
2
3



In [ ]:
```
plt.figure(figsize=(10, 8))

# Logistic Regression
mean_tpr = np.mean(log_reg_tprs, axis=0)
mean_auc = auc(mean_fpr, mean_tpr)
plt.plot(mean_fpr, mean_tpr, color='blue', label=r'Logistic Regression ROC (

# KNN
mean_tpr = np.mean(knn_tprs, axis=0)
mean_auc = auc(mean_fpr, mean_tpr)
plt.plot(mean_fpr, mean_tpr, color='red', label=r'KNN ROC (AUC = %0.2f)' % m

# Chance Diagonal
plt.plot([0, 1], [0, 1], linestyle='--', lw=2, color='black', alpha=0.8)
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC curves')
plt.legend(loc='lower right')
plt.show()
```



The purpose of using cross validation for this problem is to evaluate the performance of the model on data that was not used to train the model. This allows us to see how well the model generalizes to new data. Holding out data in validation sets keeps the model training process clean of any bias that may be introduced by using the same data for training and testing.

Comparing the models in terms of performance shows that, for this problem, logistic regression is the better model. The ROC curve for logistic regression is closer to the top left corner of the plot, which indicates a better model. The decision boundary for logistic regression is a straight line, which seems to be a better fit than the curve generated by KNN.

2

Digits classification

[30 points]

An exploration of regularization, imbalanced classes, ROC and PR curves

The goal of this exercise is to apply your supervised learning skills on a very different dataset: in this case, image data; MNIST: a collection of images of handwritten digits. Your goal is to train a classifier that is able to distinguish the number "3" from all possible numbers and to do so as accurately as possible. You will first explore your data (this should always be your starting point to gain domain knowledge about the problem.). Since the feature space in this problem is 784-dimensional, overfitting is possible. To avoid overfitting you will investigate the impact of regularization on generalization performance (test accuracy) and compare regularized and unregularized logistic regression model test error against other classification techniques such as linear discriminant analysis and random forests and draw conclusions about the best-performing model.

Start by loading your dataset from the MNIST dataset of handwritten digits, using the code provided below. MNIST has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered in a fixed-size image.

Your goal is to classify whether or not an example digit is a 3. Your binary classifier should predict $y = 1$ if the digit is a 3, and $y = 0$ otherwise. Create your dataset by transforming your labels into a binary format (3's are class 1, and all other digits are class 0).

(a) Plot 10 examples of each class (i.e. class $y = 0$, which are not 3's and class $y = 1$ which are 3's), from the training dataset.

- Note that the data are composed of samples of length 784. These represent 28 x 28 images, but have been reshaped for storage convenience. To plot digit examples, you'll need to reshape the data to be 28 x 28 (which can be done with numpy `reshape`).

(b) How many examples are present in each class? Show a plot of samples by class (bar plot). What fraction of samples are positive? What issues might this cause?

(c) Identify the value of the regularization parameter that optimizes model performance on out-of-sample data. Using a logistic regression classifier, apply lasso regularization and retrain the model and evaluate its performance on the test set over a range of values on the regularization coefficient. You can implement this using the LogisticRegression module and activating the 'l1' penalty; the parameter C is the inverse of the regularization strength. Vary the value of C logarithmically from 10^{-4} to 10^{4} (and make your x-axes logarithmic in scale) and evaluate it at least 20 different values of C. As you vary the regularization coefficient, Plot the following four quantities (this should result in 4 separate plots)...

- The number of model parameters that are estimated to be nonzero (in the logistic regression model, one attribute is `coef_`, which gives you access to the model parameters for a trained model)
- The cross entropy loss (which can be evaluated with the Scikit Learn `log_loss` function)

- Area under the ROC curve (AUC)
- The F_1-score (assuming a threshold of 0.5 on the predicted confidence scores, that is, scores above 0.5 are predicted as Class 1, otherwise Class 0). Scikit Learn also has a `f1_score` function which may be useful. -Which value of C seems best for this problem? Please select the closest power of 10. You will use this in the next part of this exercise.

(d) Train and test a (1) logistic regression classifier with minimal regularization (using the Scikit Learn package, set penalty='l1', C=1e100 to approximate this), (2) a logistic regression classifier with the best value of the regularization parameter from the last section, (3) a Linear Discriminant Analysis (LDA) Classifier, and (4) a Random Forest (RF) classifier (using default parameters for the LDA and RF classifiers).

- Compare your classifiers' performance using ROC and Precision Recall (PR) curves. For the ROC curves, all your curves should be plotted on the same set of axes so that you can directly compare them. Please do the same wih the PR curves.
- Plot the line that represents randomly guessing the class (50% of the time a "3", 50% not a "3"). You SHOULD NOT actually create random guesses. Instead, you should think through the theory behind how ROC and PR curves work and plot the appropriate lines. It's a good practice to include these in ROC and PR curve plots as a reference point.
- For PR curves, an excellent resource on how to correctly plot them can be found here (ignore the section on "non-linear interpolation between two points"). This describes how a random classifier is represented in PR curves and demonstrates that it should provide a lower bound on performance.
- When training your logistic regression model, it's recommended that you use solver="liblinear"; otherwise, your results may not converge.
- Describe the performance of the classifiers you compared. Did the regularization of the logistic regression model make much difference here? Which classifier you would select for application to unseen data.

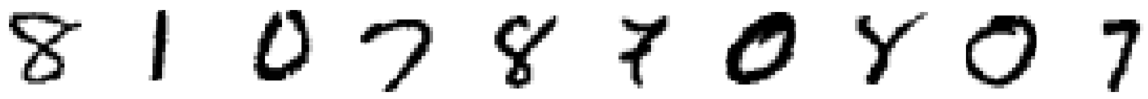
In [ ]:
```
# Load the MNIST Data
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
import pickle

# Set this to True to download the data for the first time and False after t
#   so that you just load the data locally instead
download_data = False

if download_data:
    # Load data from https://www.openml.org/d/554
    X, y = fetch_openml('mnist_784', return_X_y=True, as_frame=False)

    # Adjust the labels to be '1' if y==3, and '0' otherwise
    y[y!='3'] = 0
    y[y=='3'] = 1
    y = y.astype('int')

    # Divide the data into a training and test split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/7,

    file = open('tmpdata', 'wb')
    pickle.dump((X_train, X_test, y_train, y_test), file)
    file.close()
else:
    file = open('tmpdata', 'rb')
    X_train, X_test, y_train, y_test = pickle.load(file)
    file.close()
```

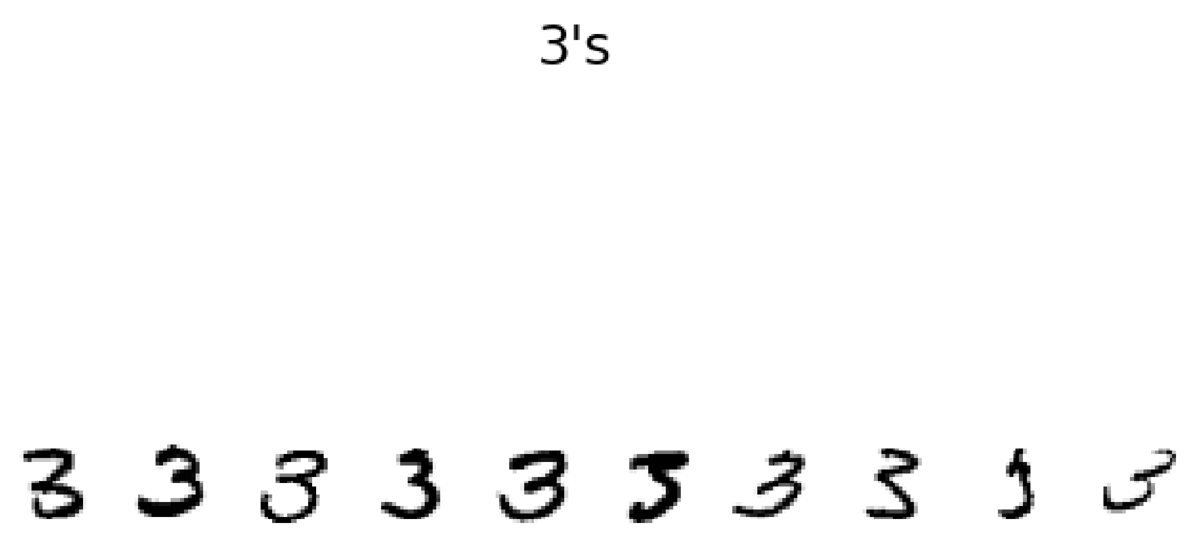
ANSWER

In [ ]:

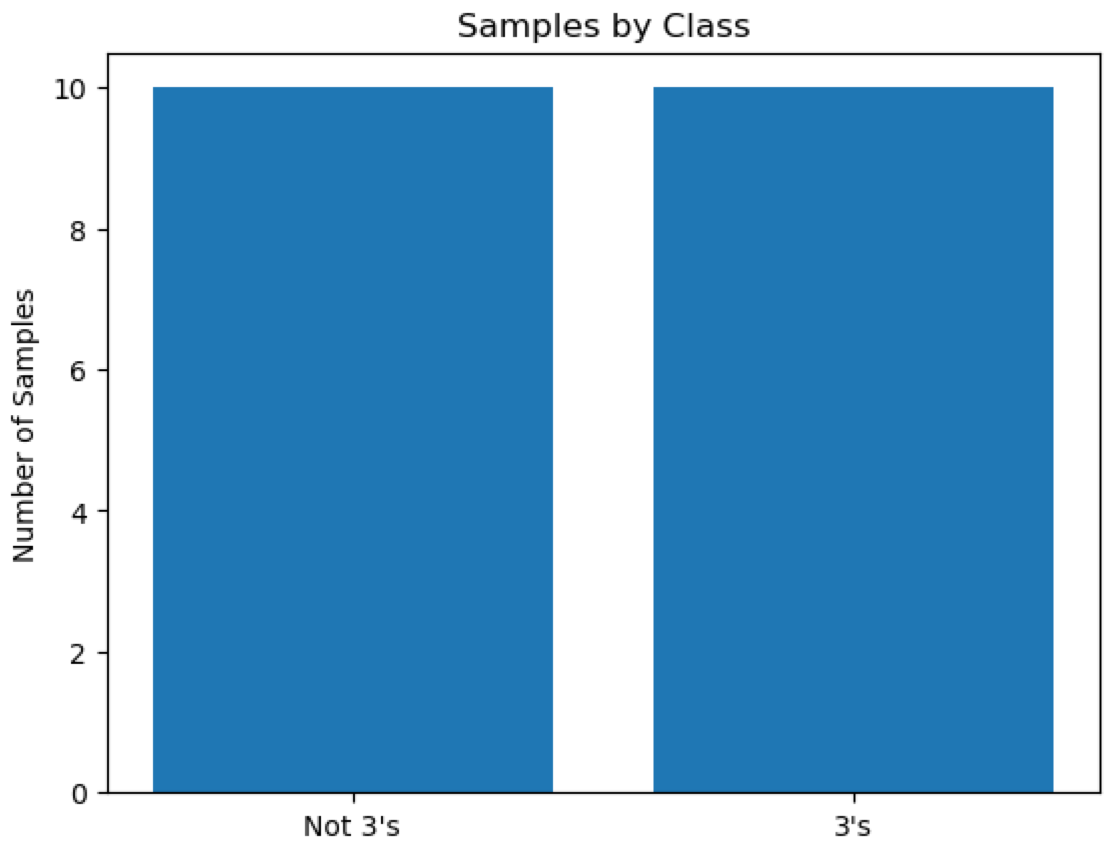
```
# a) plot 10 examples of each class from the training set
def plot_digits(data, title):
    fig, axes = plt.subplots(1, 10, figsize=(8, 5))
    for ax, image in zip(axes, data):
        ax.imshow(image.reshape(28, 28), cmap=plt.cm.gray_r, interpolation='
        ax.axis('off')
    plt.suptitle(title, size=20)
    plt.show()

# Get indices of class 0 and class 1 samples
indices_class_0 = np.where(y_train == 0)[0][:10]
indices_class_1 = np.where(y_train == 1)[0][:10]

plot_digits(X_train[indices_class_0], "Not 3's")
plot_digits(X_train[indices_class_1], "3's")
```

Not 3's



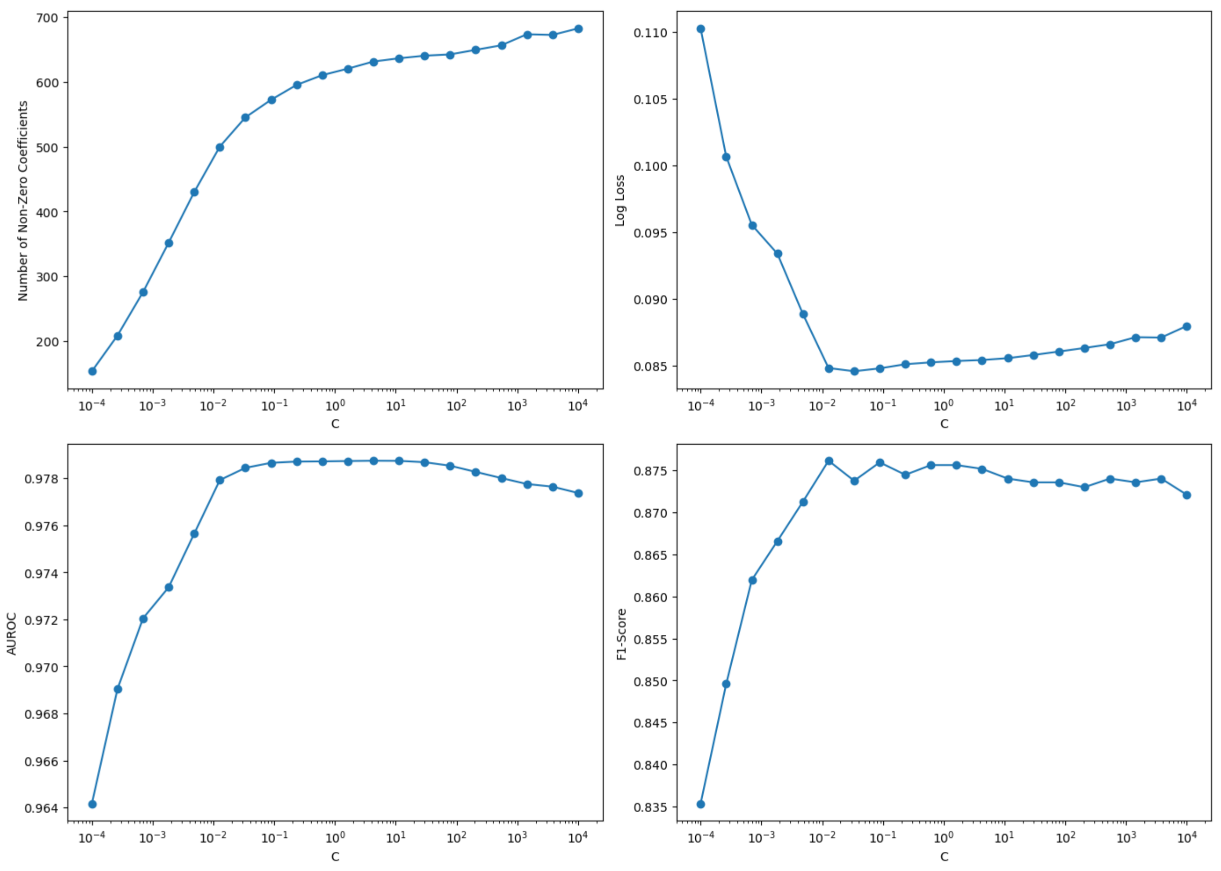


In [ ]:
```
# b) examples present in each class
plt.bar([0, 1], [len(indices_class_0), len(indices_class_1)])
plt.xticks([0, 1], ["Not 3's", "3's"])
plt.ylabel('Number of Samples')
plt.title('Samples by Class')
plt.show()

fraction_positive = len(indices_class_1) / (len(indices_class_0) + len(indic
print(f"Fraction of samples that are positive: {fraction_positive:.2f}")

print('There seems to be a nice balance of classes, but in this case that me
```



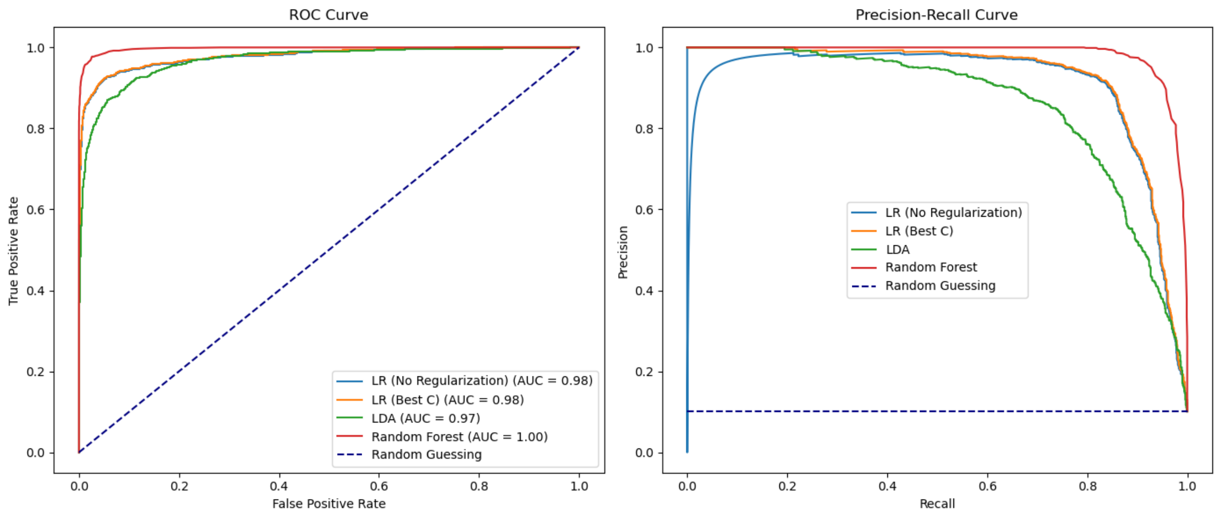
```
Fraction of samples that are positive: 0.50
There seems to be a nice balance of classes, but in this case that means the
dataset is half threes, half other values, which is sort of an imbalance, ev
en if it is a binary class situation.
```

In [ ]:
```python
# c) optimize regularization parameter

# Define values of C
C_values = np.logspace(-4, 4, 20)

non_zero_coefs = [] # number of non-zero coefficients
log_losses = [] # log loss
aurocs = [] # area under the ROC curve
f1_scores = [] # F1 score

for C in C_values:
    lr = LogisticRegression(penalty='l1', C=C, solver="liblinear")
    lr.fit(X_train, y_train)
    y_pred_proba = lr.predict_proba(X_test)[:, 1]

    non_zero_coefs.append(np.sum(lr.coef_ != 0))
    log_losses.append(log_loss(y_test, y_pred_proba))
    aurocs.append(roc_auc_score(y_test, y_pred_proba))
    y_pred = np.where(y_pred_proba > 0.5, 1, 0)
    f1_scores.append(f1_score(y_test, y_pred))

# Plot the results
fig, axarr = plt.subplots(2, 2, figsize=(14, 10))

# Plot non-zero coefficients
axarr[0, 0].semilogx(C_values, non_zero_coefs, marker='o')
axarr[0, 0].set_xlabel('C')
axarr[0, 0].set_ylabel('Number of Non-Zero Coefficients')

# Plot log-loss
axarr[0, 1].semilogx(C_values, log_losses, marker='o')
axarr[0, 1].set_xlabel('C')
axarr[0, 1].set_ylabel('Log Loss')

# Plot AUROC
axarr[1, 0].semilogx(C_values, aurocs, marker='o')
axarr[1, 0].set_xlabel('C')
axarr[1, 0].set_ylabel('AUROC')

# Plot F1-Score
axarr[1, 1].semilogx(C_values, f1_scores, marker='o')
axarr[1, 1].set_xlabel('C')
axarr[1, 1].set_ylabel('F1-Score')

plt.tight_layout()
plt.show()

# Identify and print the best C
best_C = C_values[np.argmax(aurocs)]
print(f"The best value of C is approximately: {best_C}")
```



```
The best value of C is approximately: 4.281332398719396
```

In [ ]:
```
# d) train classifiers and compare

# Define and train the classifiers
classifiers = {
    "LR (No Regularization)": LogisticRegression(penalty='l1', C=1e100, solv
    "LR (Best C)": LogisticRegression(penalty='l1', C=best_C, solver="liblir
    "LDA": LinearDiscriminantAnalysis(),
    "Random Forest": RandomForestClassifier()
}

plt.figure(figsize=(14, 6))
for name, clf in classifiers.items():
    clf.fit(X_train, y_train)
    y_pred_proba = clf.predict_proba(X_test)[:, 1]

    # ROC curve
    fpr, tpr, _ = roc_curve(y_test, y_pred_proba)
    plt.subplot(1, 2, 1)
    plt.plot(fpr, tpr, label=f"{name} (AUC = {auc(fpr, tpr):.2f})")

    # PR curve
    precision, recall, _ = precision_recall_curve(y_test, y_pred_proba)
    plt.subplot(1, 2, 2)
    plt.plot(recall, precision, label=f"{name}")

# ROC curve settings
plt.subplot(1, 2, 1)
plt.plot([0, 1], [0, 1], color='navy', linestyle='--', label="Random Guessir
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend()

# PR curve settings
plt.subplot(1, 2, 2)
no_skill = len(y_test[y_test==1]) / len(y_test) # calculate the no skill lin
plt.plot([0, 1], [no_skill, no_skill], color='navy', linestyle='--', label="
plt.xlabel('Recall')
plt.ylabel('Precision')
plt.title('Precision-Recall Curve')
plt.legend()

plt.tight_layout()
plt.show()

# describe and compare the classifier's performance
```



The graphs above show the results of training 4 different classifiers against the MNIST dataset. The first graph shows the ROC curves for each classifier, and the second graph shows the PR curves for each classifier. As can be seen, the LR models performed very similarly to each other. The LDA model was slightly les effective. Random Forest is shown here to be the most effective classifier, with the highest AUC and PR curve scores (hugging the corner of the graph). I would select the random forest classfier to take advantage of the strength of ensemble models.