Assignment 3 - Supervised Learning: model training and evaluation

Fides Regina Schwartz

Netid: fs113

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

Question 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).

I. Load, prepare and plot the data

(a) i) Load the data

In [ ]:
```
import pandas as pd

Q1 = pd.read_csv(
    "C:\\Users\\dm93\\Documents\\GitHub\\ids705\\assignments\\data\\A3_Q1_data.csv",
    header=0,
)
```

In [ ]:
```
# Look at data
Q1.head()
```

Out[ ]:

	x1	x2	y
0	2.553124	0.337757	0
1	-0.877757	0.045790	1
2	-0.903528	0.368076	1
3	-1.532152	-0.863737	1
4	-0.046954	-0.388761	1

Header has been chosen correctly

In [ ]:
```
# Look at shape of data
Q1.shape
```

Out[ ]:
```
(200, 3)
```

We have 200 rows and 3 columns of data

In [ ]:
```
# Look at type of data
type(Q1)
```

Out[ ]:
```
pandas.core.frame.DataFrame
```

Data format is pandas dataframe

In [ ]:
```
# Check for missing values
Q1.isnull().values.any()
```

Out[ ]:
```
False
```

There are no missing values

In [ ]:
```
# Look at summary stats
Q1.describe()
```

Out[ ]:

	x1	x2	y
count	200.000000	200.000000	200.000000
mean	0.151376	-0.385426	0.485000
std	1.411722	1.217490	0.501029
min	-3.210005	-3.193456	0.000000
25%	-0.912029	-1.341047	0.000000
50%	0.112286	-0.479684	0.000000
75%	1.174400	0.495114	1.000000
max	3.867647	3.103541	1.000000

All columns have 200 rows, the minimum and maximum values for x1 and x2 seem to be in roughly the same range (-3.5 to 4). The mean value of y is ~0.5, so it seems like there's an equal distribution of 0 and 1 classes.

ii) 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.

In [ ]:
```
# import everything sklearn
import sklearn
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.linear_model import LinearRegression
from sklearn import metrics
```

```
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_openml
```

In [ ]:
```
# Split data into X and y sets
X = Q1.iloc[:, [0,1]]
y = Q1.iloc[:, -1]
```

In [ ]:
```
# Check that the split worked
#print(y.head())
print(X.head())
```

```
         x1        x2
0  2.553124  0.337757
1 -0.877757  0.045790
2 -0.903528  0.368076
3 -1.532152 -0.863737
4 -0.046954 -0.388761
```

In [ ]:
```
# Use sklearn to split into train and test dataset
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

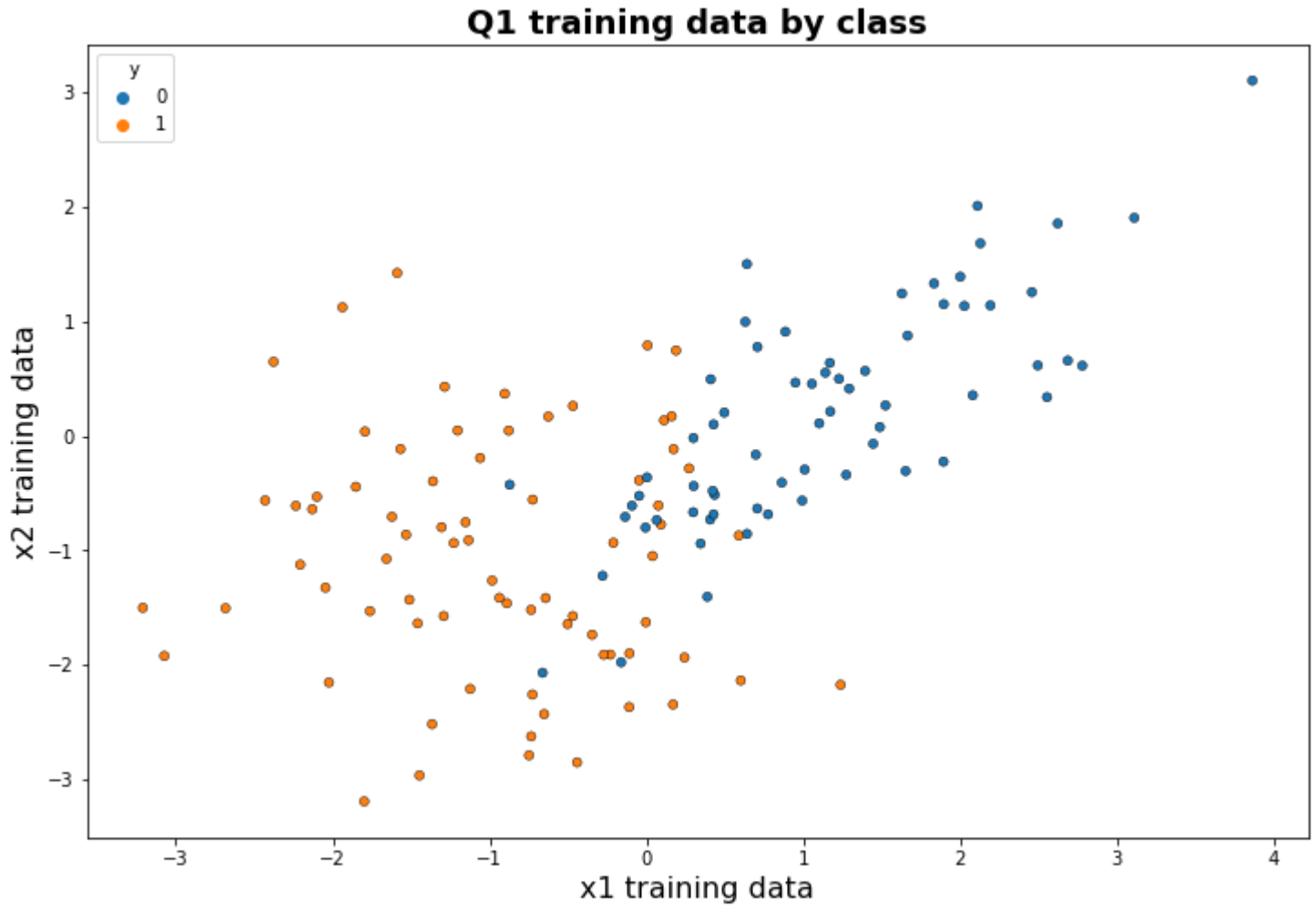
In [ ]:
```
type(X_train)
```

Out[ ]:
```
pandas.core.frame.DataFrame
```

iii) Plot the training data by class

In [ ]:
```
# import all things plotting
import numpy as np
import scipy as sp
import scipy.spatial
import seaborn as sns
import pickle
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
%matplotlib inline
```

In [ ]:
```
fig, ax = plt.subplots(figsize=(12,8))

sns.scatterplot(
    x=X_train.iloc[:, 0], y=X_train.iloc[:, 1], hue=y, marker="o", s=25, edgecolor="k", ax=ax
).set_title("Q1 training data by class", fontsize=18, fontweight="bold")
ax.set_xlabel("x1 training data", fontsize=16)
ax.set_ylabel("x2 training data", fontsize=16)
plt.show()
```



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

The data does not look completely separable as there is some overlap between the classes around the x1 and x2 0 variables. A logistic regression would probably be able to produce reasonable predictions though, because the decision boundary could be visualized as a

diagonal between the two classes and mostly separate the classes.

(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?

There are no missing values in the dataset and the minimum and maximum values for x1 and x2 seem to be in roughly the same range (-3.5 to 4). The mean value of y is ~0.5, so it seems like there's an equal distribution of 0 and 1 classes.

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 $\sigma(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}$. 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 can be used 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* $\{(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \ldots, (y_N, \mathbf{x}_N)\}$.

Sigmoid function:

$$\begin{aligned} L(w|y, X) = P(y_1, y_2, \ldots, y_N | x_1, x_2, \ldots, x_N) \\ &= \prod_{i=1}^{N} P(y_i|x_i) \\ &= \prod_{i=1}^{N} [\sigma(w^\top x_i)]^{y_i} [1 - \sigma(w^\top x_i)]^{1-y_i} \end{aligned}$$

Likelihood function for entire training dataset:

$$\boxed{L(w|y, X) = \prod_{i=1}^{N} [\sigma(w^\top x_i)]^{y_i} [1 - \sigma(w^\top x_i)]^{1-y_i}}$$

(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 likelihood as a cost function by taking negative log on both sides of the equation:

$C(\mathbf{w}) = -log\left[\prod_{i=1}^{N} [\sigma(w^\top x_i)]^{y_i} [1 - \sigma(w^\top x_i)]^{1-y_i}\right]$

$= -\left[\sum_{\mathbf{i=1}}^{\mathbf{N}} \log[\sigma(\mathbf{w}^\top \mathbf{x_i})]^{\mathbf{y_i}} [\mathbf{1} - \sigma(\mathbf{w}^\top \mathbf{x_i})]^{\mathbf{1-y_i}}\right]$

$= -\left[\sum_{\mathbf{i=1}}^{\mathbf{N}} \mathbf{y_i} \log(\sigma(\mathbf{w}^\top \mathbf{x_i})) + \mathbf{(1 - y_i) log}[\mathbf{1} - \sigma(\mathbf{w}^\top \mathbf{x_i}))\right]$

Define the cost function:

$$\boxed{C(\mathbf{w}) = -\left[\sum_{i=1}^{N} y_i \log(\sigma(w^\top x_i)) + (1 - y_i) \log(1 - \sigma(w^\top x_i))\right]}$$

The average cost function (normalized to sample size) is:

$$\boxed{C(\mathbf{w}) = -\frac{1}{N}\left[\sum_{i=1}^{N} y_i \log(\sigma(w^\top x_i)) + (1-y_i)\log(1-\sigma(w^\top x_i))\right]}$$

(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. $\nabla_{\mathbf{w}} C(\mathbf{w}) = \left[\frac{\partial C}{\partial w_0}, \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}\right]$.

First, take the partial derivatives of each weight:

$$\begin{aligned}\frac{\partial C}{\partial w_0} &= \sum_{i=1}^{N} x_{i,0}[y_i(1-\sigma(w^\top x_i)) - (1-y_i)\sigma(w^\top x_i)] \\ &= \sum_{i=1}^{N} x_{i,0}[\sigma(w^\top x_i) - y_i]\end{aligned}$$

$$\begin{aligned}\frac{\partial C}{\partial w_1} &= \sum_{i=1}^{N} x_{i,1}[y_i(1-\sigma(w^\top x_i)) - (1-y_i)\sigma(w^\top x_i)] \\ &= \sum_{i=1}^{N} x_{i,1}[\sigma(w^\top x_i) - y_i]\end{aligned}$$

$$\begin{aligned}\frac{\partial C}{\partial w_2} &= \sum_{i=1}^{N} x_{i,2}[y_i(1-\sigma(w^\top x_i)) - (1-y_i)\sigma(w^\top x_i)] \\ &= \sum_{i=1}^{N} x_{i,2}[\sigma(w^\top x_i) - y_i]\end{aligned}$$

Second, express them all together like this:

$$\begin{aligned}\nabla_{\mathbf{w}} C(\mathbf{w}) &= \left[\frac{\partial C}{\partial w_0}, \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}\right] \\ &= \left[\sum_{i=1}^{N} x_{i,0}[\sigma(w^\top x_i) - y_i], \sum_{i=1}^{N} x_{i,1}[\sigma(w^\top x_i) - y_i], \sum_{i=1}^{N} x_{i,2}[\sigma(w^\top x_i) - y_i]\right]\end{aligned}$$

Third, combine into one step:

$$\nabla_{\mathbf{w}} C(\mathbf{w}) = \left[\sum_{i=1}^{N} [\sigma(w^\top x_i) - y_i], \sum_{i=1}^{N} x_] i, 1[\sigma(w^\top x_i) - y_i], \sum_{i=1}^{N} x_{i,2} [\sigma(w^\top x_i) - y_i] \right]$$

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.

The following are the gradient descent update functions for each w_i:

$$w_0^{(k+1)} = w_0^{(k)} - \eta \frac{\partial C}{\partial w_0^{(k)}}$$

$$w_1^{(k+1)} = w_1^{(k)} - \eta \frac{\partial C}{\partial w_1^{(k)}}$$

$$w_2^{(k+1)} = w_2^{(k)} - \eta \frac{\partial C}{\partial w_2^{(k)}}$$

If we want to generalize this, it looks like this:

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

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.).

In [ ]:
```
y.shape
type(y_train)
```

```
pandas.core.series.Series
```

Out[ ]:

In [ ]:
```
#y = y.values
y_train = y_train.values
```

In [ ]:
```
import random

# 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 weights
        #  at each iteration of gradient descent to build our
        #  learning curves
        # returns nothing
        pass

    # Method for calculating the sigmoid function of w^T X for an input set of weights
    def sigmoid(self, X, w):
        sigma = 1 / (1 + np.exp(-w.reshape(1, -1) @ X.T)).T  # to get Nx1
        return sigma  # returns the value of the sigmoid

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

    # Update the weights in an iteration of gradient descent
    def gradient_descent(self, X, y, w, lr):
        pred = self.sigmoid(X, w)
        #print(y)
        y = y.reshape(-1, 1) # make y a column vector
        d = lr * np.sum(np.dot(X.T, (pred-y)))
        #d = y-pred.T
        #print(d.shape)
        #d@X
        #summing = X.T@d
        #print(self.w.shape)
        #print(summing.shape)
```

```
        #print(X.shape)
        #print(lr)
        #self.w = self.w + lr*summing
        self.w -= d
        delta = np.linalg.norm(d) # updated weights
        # returns s scalar of the magnitude of the Euclidean norm
        #  of the change in the weights during one gradient descent step
        return delta

    # Fit the logistic regression model to the data through gradient descent
    def fit(self, X, y, w, lr, delta_thresh=1e-6, max_iter=5000, verbose=False):
        # Note the verbose flag enables you to print out the weights at each iteration
        #  (optional - but may help with one of the questions)
        self.w = w
        X = self.prepare_x(X)
        iterations = 0
        update_norm = 1

        while (iterations < max_iter) and (update_norm > delta_thresh):
            update_vec = self.gradient_descent(X, y, w, lr)
            w_init[:, None] -= update_vec
            self.w = w_init
            self.saved_w.append(w_init.copy())
            iterations += 1
            if verbose:
                print(self.w)  # returns nothing

    # Use the trained model to predict the confidence scores (prob of positive class in this case)
    def predict_proba(self, X):
        return self.sigmoid(
            X, self.w
        )  # returns the confidence score for the each sample

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

    # Stores the learning curves from saved weights from gradient descent
    def learning_curve(self, X, y):
        lr = []
        for i in range(0, len(self.saved_w)):
            pass  # returns the value of the cost function from each step in gradient descent
        #  from the last model fitting process
```

```
        return lr

    # Appends a column of ones as the first feature to account for the bias term
    def prepare_x(self, X):
        return np.concatenate([np.ones((X.shape[0],1)), X], axis = 1) # returns the X with a new feature of all ones (a c
```

(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 $\{10^{-2}, 10^{-4}, 10^{-6}\}$. 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 for direct comparison. *Note: make sure you're using average cost per sample, not total cost*

- Try running this process for a really big learning rate for this problem: 10^{0}. Look at the weights that the fitting process generates over the first 50 iterations and how they change. You may simply print these first 50 iterations as output or plot them. What happens and why?
- What is the impact that the different values of learning has 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.

h.1 Choose a learning rate and fit your model. Run this process for a really big learning rate for this problem: 10^{0}. Look at the weights that the fitting process generates over the first 50 iterations and how they change. You may simply print these first 50 iterations as output or plot them. What happens and why?

In [ ]:
```
model1 = Logistic_regression()
w_init = np.array([random.random() for i in range(3)])
model1.fit(X_train, y_train, w_init, lr = 0.01, verbose=False)
model1.saved_w
#model1.learning_curve(X_train, y_train)
X = model1.prepare_x(X_train)
#np.array([random.random() for i in range(X.shape[1])])
#print(X)
```

h.2 Show the learning curves for the gradient descent process for learning rates of $\{10^{-2}, 10^{-4}, 10^{-6}\}$. 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.

h.3 What is the impact that the different values of learning has on the speed of the process and the results?

h.4 Of the options explored, what learning rate do you prefer and why?

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 two plots corresponding to each model (logistic regression and KNN): one with the training data, and one for 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?

i.1 Train log regression model and a KNN with k = 7.

In [ ]:
```
X_train.to_numpy()
X_test.to_numpy()
```

In [ ]:
```
type(X_train)
```

Out[ ]:
```
pandas.core.frame.DataFrame
```

In [ ]:
```
#Train logistic regression model
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0).fit(X_train, y_train)
```

```
clf.predict(X_train.iloc[:2, :])

clf.predict_proba(X_train.iloc[:2, :])


clf.score(X_train, y_train)
```

Out[ ]:
```
0.8642857142857143
```

In [ ]:
```
# Implement KNN using sklearn for training data
knn_train = KNeighborsClassifier(n_neighbors=7)

# Fit KNN
knn_train.fit(X_train, y_train)

# Predict
y_hat_train = knn_train.predict(X_train)
```

In [ ]:
```
# Implement KNN using sklearn for test data
knn_test = KNeighborsClassifier(n_neighbors=7)

# Fit KNN
knn_test.fit(X_test, y_test)

# Predict
y_hat_test = knn_test.predict(X_test)
```

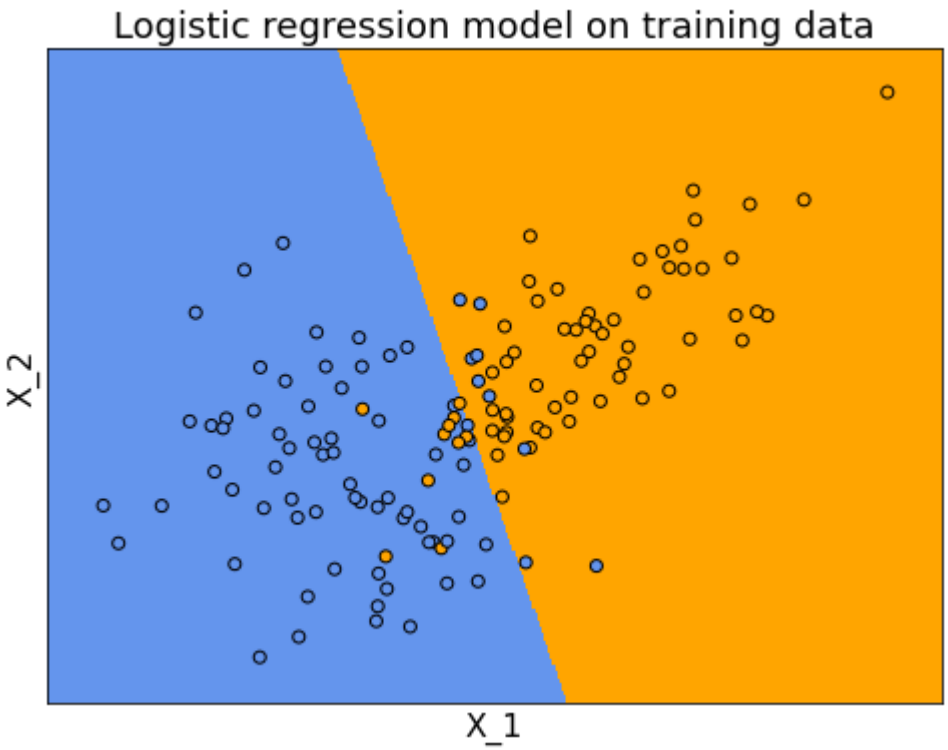
i.2 Make two plots for each model (test and train each) with decision boundary (4 plots total)

Plot 1 - Logistic regression on training data

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.linear_model import LogisticRegression
from sklearn import datasets

h = 0.02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(["orange", "cyan", "cornflowerblue"])
```

```
cmap_bold = ["darkorange", "darkblue"]

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X_train.iloc[:, 0].min() - 0.5, X_train.iloc[:, 0].max() + 0.5
y_min, y_max = X_train.iloc[:, 1].min() - 0.5, X_train.iloc[:, 1].max() + 0.5

xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure(1, figsize=(8, 6))
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)

# Plot also the training points
plt.scatter(
    X_train.iloc[:, 0], X_train.iloc[:, 1], c=y_train, edgecolor="black", cmap=cmap_light
)
plt.xlabel("X_1", fontsize=16)
plt.ylabel("X_2", fontsize=16)
plt.title("Logistic regression model on training data", fontdict={"fontsize":18})
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xticks(())
plt.yticks(())

plt.show()
```

```
C:\Users\dm93\miniconda3\lib\site-packages\sklearn\base.py:445: UserWarning: X does not have valid feature names, but Log
isticRegression was fitted with feature names
  warnings.warn(
```



Plot 2 - Logistic regression on test data

In [ ]:

```
h = 0.02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(["orange", "cyan", "cornflowerblue"])
cmap_bold = ["darkorange", "darkblue"]

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X_test.iloc[:, 0].min() - 0.5, X_test.iloc[:, 0].max() + 0.5
y_min, y_max = X_test.iloc[:, 1].min() - 0.5, X_test.iloc[:, 1].max() + 0.5

xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure(1, figsize=(8, 6))
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)
```

```
plt.contourf(xx, yy, Z, cmap=cmap_light)

# Plot also the training points
plt.scatter(
    X_test.iloc[:, 0], X_test.iloc[:, 1], c=y_test, edgecolor="black", cmap=cmap_light
)
plt.xlabel("X_1", fontsize=16)
plt.ylabel("X_2", fontsize=16)
plt.title("Logistic regression model on test data", fontdict={"fontsize":18})

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.xticks(())
plt.yticks(())

plt.show()
```

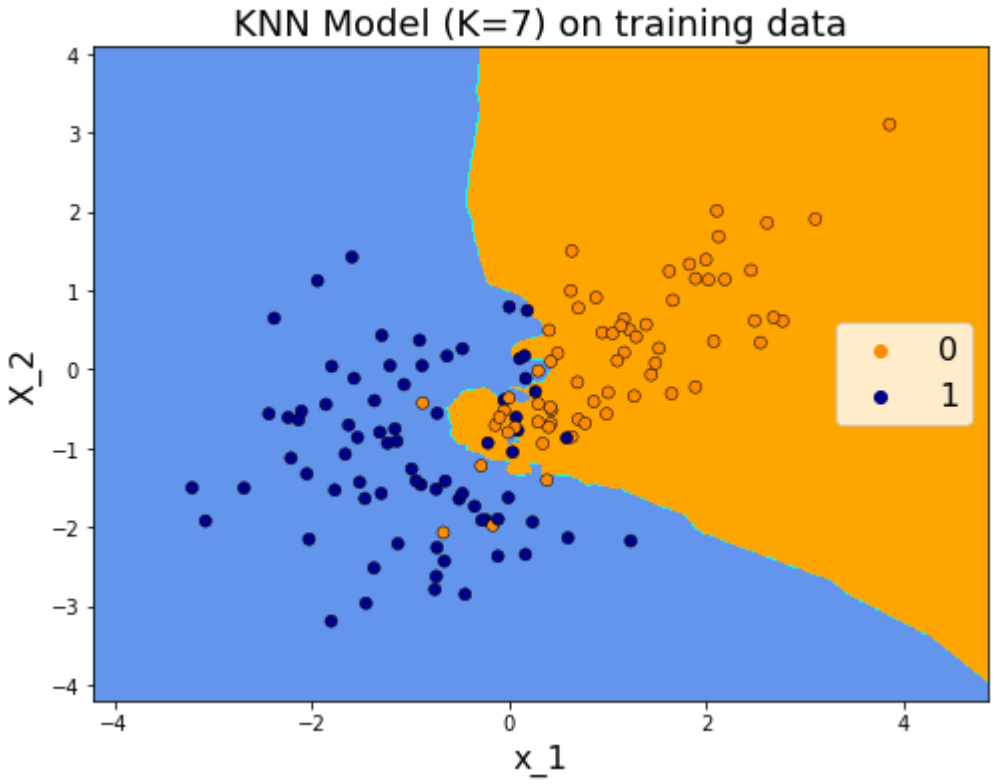
```
C:\Users\dm93\miniconda3\lib\site-packages\sklearn\base.py:445: UserWarning: X does not have valid feature names, but Log
isticRegression was fitted with feature names
  warnings.warn(
```

Plot 3 - KNN (K=7) on training data

In [ ]:

```
# nearest neighbour = 7

h = 0.02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(["orange", "cyan", "cornflowerblue"])
cmap_bold = ["darkorange", "darkblue"]

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X_train.iloc[:, 0].min() - 1, X_train.iloc[:, 0].max() + 1
y_min, y_max = X_train.iloc[:, 1].min() - 1, X_train.iloc[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = knn_train.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure(figsize=(8, 6))
plt.contourf(xx, yy, Z, cmap=cmap_light)

# Plot also the training points
sns.scatterplot(
    x=X_train.iloc[:, 0],
    y=X_train.iloc[:, 1],
    hue=y_train,
    palette=cmap_bold,
    alpha=1.0,
    edgecolor="black",
)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("KNN Model (K=7) on training data", fontsize=18)
plt.xlabel("x_1", fontsize=16)
plt.ylabel("X_2", fontsize=16)
plt.legend(loc="right", fontsize=16)
plt.show()
```

```
C:\Users\dm93\miniconda3\lib\site-packages\sklearn\base.py:445: UserWarning: X does not have valid feature names, but KNe
ighborsClassifier was fitted with feature names
  warnings.warn(
```



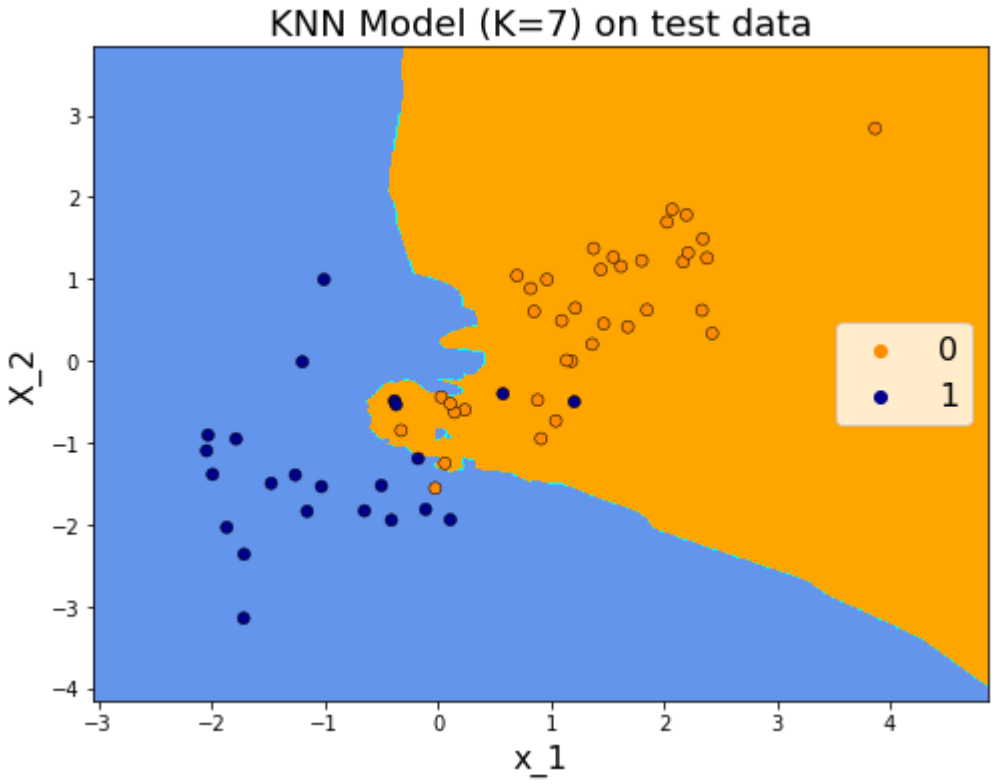
Plot 4 - KNN (K=7) on test data

In [ ]:

```
h = 0.02  # step size in the mesh

# Create color maps
cmap_light = ListedColormap(["orange", "cyan", "cornflowerblue"])
cmap_bold = ["darkorange", "darkblue"]

# Plot the decision boundary. For that, we will assign a color to each
# point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X_test.iloc[:, 0].min() - 1, X_test.iloc[:, 0].max() + 1
y_min, y_max = X_test.iloc[:, 1].min() - 1, X_test.iloc[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = knn_train.predict(np.c_[xx.ravel(), yy.ravel()])

# Put the result into a color plot
Z = Z.reshape(xx.shape)
plt.figure(figsize=(8, 6))
plt.contourf(xx, yy, Z, cmap=cmap_light)
```

```
# Plot also the training points
sns.scatterplot(
    x=X_test.iloc[:, 0],
    y=X_test.iloc[:, 1],
    hue=y_test,
    palette=cmap_bold,
    alpha=1.0,
    edgecolor="black",
)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
plt.title("KNN Model (K=7) on test data", fontsize=18)
plt.xlabel("x_1", fontsize=16)
plt.ylabel("X_2", fontsize=16)
plt.legend(loc="right", fontsize=16)
plt.show()
```

```
C:\Users\dm93\miniconda3\lib\site-packages\sklearn\base.py:445: UserWarning: X does not have valid feature names, but KNe
ighborsClassifier was fitted with feature names
  warnings.warn(
```



i.3 Produce ROC curve, use k = 10 folds for cross validation, plot on the same axes + chance diagonal + AUC in legend

In [ ]:

```
from sklearn.metrics import roc_curve, auc
from sklearn.model_selection import StratifiedKFold

# k = 10 folds cross validation with ROC curves
cv = StratifiedKFold(n_splits=10)

# set up arrays for scores to go into KNN
y_scores_knn = np.array([])
y_pred_knn = np.array([])
y_true_knn = np.array([])

# set up arrays for scores to go into logistic regression
y_scores_logreg = np.array([])
y_pred_logreg = np.array([])
y_true_logreg = np.array([])

# Make predictions for each of the folds
```

```
for i, (train_index, test_index) in enumerate(cv.split(X, y)):
    X_train_cv = X[train_index]
    y_train_cv = y[train_index]
    X_test_cv = X[test_index]
    y_test_cv = y[test_index]

    # Fit KNN on training data as before
    knn = KNeighborsClassifier(n_neighbors=7, n_jobs=-1)
    knn.fit(X_train_cv, y_train_cv)
    y_scores_knn = np.concatenate((y_scores_knn, knn.predict_proba(X_test_cv)[:, 1]))
    y_pred_knn = np.concatenate((y_pred_knn, knn.predict(X_test_cv)))
    y_true_knn = np.concatenate((y_true_knn, y_test_cv))

    # Fit the logistic regression model as before
    logreg = Logistic_regression()
    x_train_cv = logreg.prepare_x(X_train_cv)
    x_test_cv = logreg.prepare_x(X_test_cv)
    logreg.fit(x_train_cv, y_train_cv, lr=10 ** -2)
    y_scores_logreg = np.concatenate(
        y_scores_logreg, logreg.predict_proba(x_test_cv).ravel()
    )
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp/ipykernel_24488/2038506030.py in <module>
     16
     17 # Make predictions for each of the folds
---> 18 for i, (train_index, test_index) in enumerate(cv.split(X, y)):
     19     X_train_cv = X[train_index]
     20     y_train_cv = y[train_index]

~\miniconda3\lib\site-packages\sklearn\model_selection\_split.py in split(self, X, y, groups)
    328             The testing set indices for that split.
    329         """
--> 330         X, y, groups = indexable(X, y, groups)
    331         n_samples = _num_samples(X)
    332         if self.n_splits > n_samples:

~\miniconda3\lib\site-packages\sklearn\utils\validation.py in indexable(*iterables)
    368     """
    369     result = [_make_indexable(X) for X in iterables]
--> 370     check_consistent_length(*result)
    371     return result
    372
```

```
~\miniconda3\lib\site-packages\sklearn\utils\validation.py in check_consistent_length(*arrays)
    329     uniques = np.unique(lengths)
    330     if len(uniques) > 1:
--> 331         raise ValueError(
    332             "Found input variables with inconsistent numbers of samples: %r"
    333             % [int(l) for l in lengths]

ValueError: Found input variables with inconsistent numbers of samples: [140, 200]
```

i.4 What is the purpose of cross validation for this? How do models compare in terms of performance and which model would you select for this problem, and why?

Based on the plots I produced, it looks like the KNN produces a slightly better class separation for the test set with only one point of class 0 mis-classified and four points of class 1 misclassified, while the logistic regression misclassified four points of class 0 and two points of class 1.

Question 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) Using a logistic regression classifier, apply lasso regularization and retrain the model and evaluate its performance 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 20 different values of C. As you vary the regularization coefficient, Plot

- 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.
- 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 the first time
#   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 intro a training and test split
    Mnist_X_train, Mnist_X_test, Mnist_y_train, Mnist_y_test = train_test_split(X, y, test_size=1/7, random_state=88)

    file = open('tmpdata', 'wb')
    pickle.dump((Mnist_X_train, Mnist_X_test, Mnist_y_train, Mnist_y_test), file)
    file.close()
else:
    file = open('tmpdata', 'rb')
    Mnist_X_train, Mnist_X_test, Mnist_y_train, Mnist_y_test = pickle.load(file)
    file.close()
```

In [ ]:
```
# Look at data structure
print(Mnist_X_train.shape)
print(Mnist_X_test.shape)
print(type(Mnist_X_train))
print (type(Mnist_y_train))
```

```
(60000, 784)
(10000, 784)
```

```
<class 'numpy.ndarray'>
<class 'numpy.ndarray'>
```

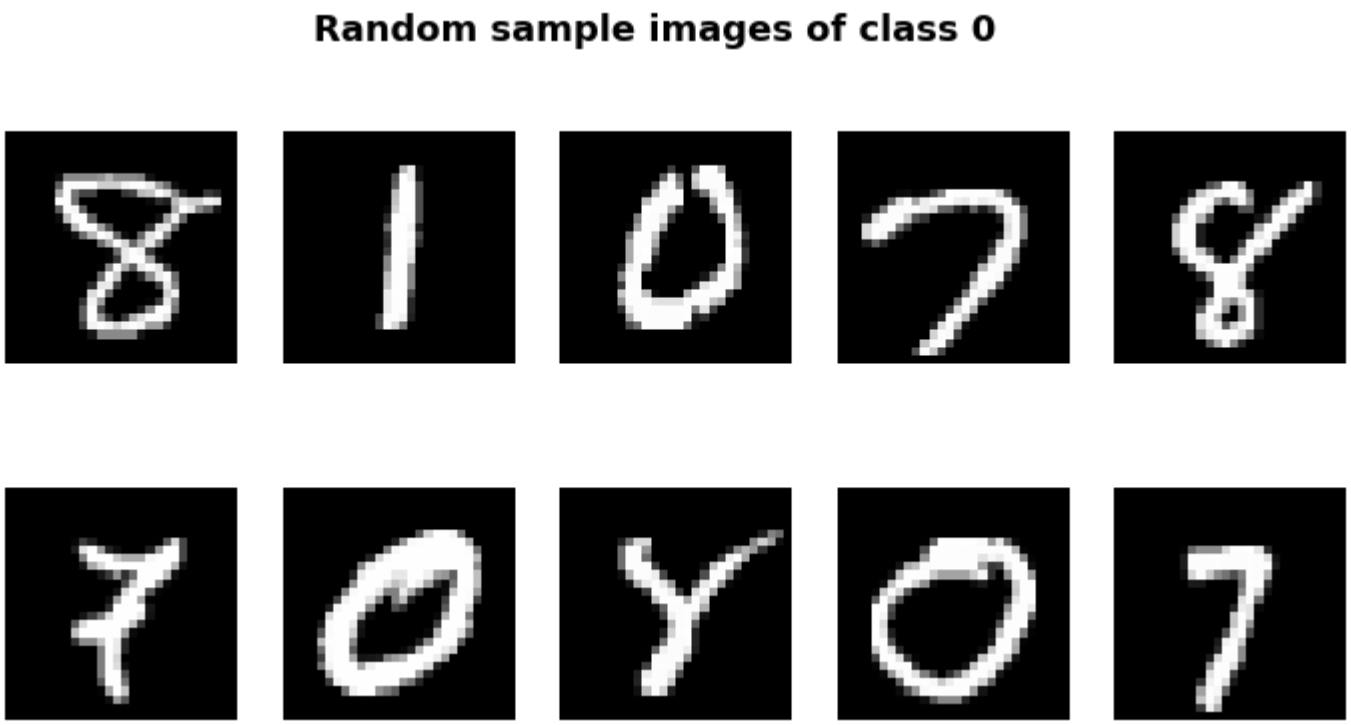
(a) Plot 10 examples of each class from the training dataset

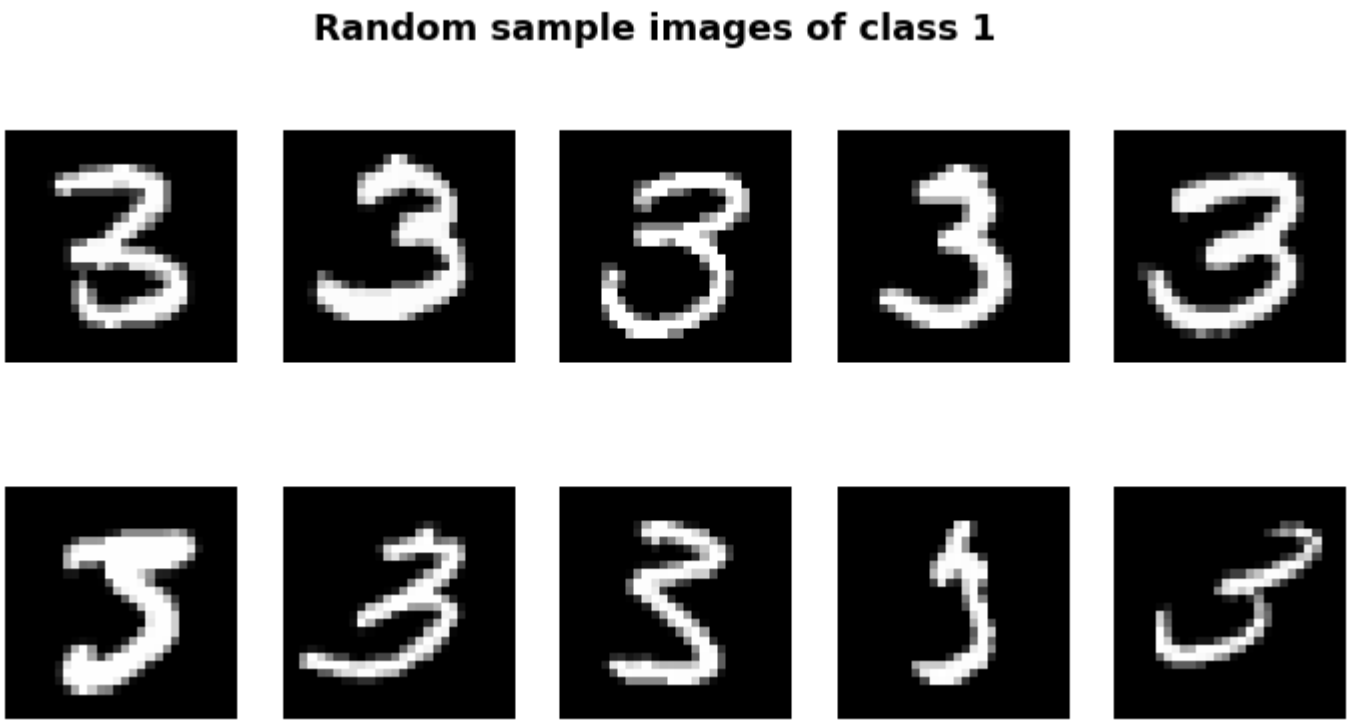
In [ ]:

```
# reshape X test and train data to 28 x 28
Mnist_X_train1 = Mnist_X_train.reshape(Mnist_X_train.shape[0], 28, 28)
Mnist_X_test1 = Mnist_X_test.reshape(Mnist_X_test.shape[0], 28, 28)

#plot 10 random images of class 0 from the training set
fig, ax = plt.subplots(2, 5, figsize=(12, 6))
for i in range(10):
    ax[i // 5, i % 5].imshow(Mnist_X_train1[Mnist_y_train == 0][i], cmap='gray')
    ax[i // 5, i % 5].axis("off")
fig.suptitle("Random sample images of class 0", fontsize=18, fontweight="bold")

#plot 10 random images of class 1 from the training set
fig, ax = plt.subplots(2, 5, figsize=(12, 6))
for i in range(10):
    ax[i // 5, i % 5].imshow(Mnist_X_train1[Mnist_y_train == 1][i], cmap='gray')
    ax[i // 5, i % 5].axis("off")
fig.suptitle("Random sample images of class 1", fontsize=18, fontweight="bold")
```

Out[ ]:

```
Text(0.5, 0.98, 'Random sample images of class 1')
```


Random sample images of class 0



(b) How many examples are present in each class? Show bar plot by class

In [ ]:
```
print("Class 0 has {nclass0} examples in the training data set")
print("Class 1 has {nclass1} examples in the training data set")
```

In [ ]:
```
# bar plot by class

fig, ax = plt.subplots(figsize=(12, 6))
sns.countplot(y_train, ax=ax)
ax.text(
    1, nclass1 - 3000, nclass1, fontsize=16, ha="center", va="center", color="white"
)
ax.text(
    0, nclass0 - 3000, nclass1, fontsize=16, ha="center", va="center", color="white"
)
ax.set_title("Number of examples per class", fontsize=16, fontweight="bold")
ax.set_xlabel("Class", fontsize=14)
ax.set_ylabel("No Examples", fontsize=14)
sns.despine()
```

What fraction of samples are positive?

In [ ]:
```
print(f"The proportion of positive samples in the training data set is {nclass1/nclass0+nclass1):.2%}")
```

What issues might this cause?

The training data is highly imbalanced between the classes (only 10% of data is positive). Several issues arise from this:

1. Accuracy can not be trusted, because extreme cases would result in high accuracy (e.g. only 0 predicted would reach 90% accuracy).
2. Overall weaker classification power due to low positive signal.

(c) Using a log regression classifier, apply lasso regularization

c.1 Plot 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)

c.2 Plot the cross entropy loss (which can be evaluated with the Scikit Learn `log_loss` function)

c.3 Plot the area under the ROC curve (AUC)

c.4 Plot 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.

c.5 Plot 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 four different classifiers

(1) logistic regression classifier with minimal regularization; When training your logistic regression model, it's recommended that you use solver="liblinear"; otherwise your results may not converge

In [ ]:
```
# import kits
from sklearn.ensemble import RandomForestClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis

# fit model log regression with min reg
logistic_min = LogisticRegression(
    random_state=2018, penalty="l1", solver="liblinear", C=1e100
).fit(Mnist_X_train, Mnist_y_train)

# define metrics
y_pred_proba_min = logistic_min.predict_proba(Mnist_X_test)[::, 1]
```

```
fpr_min, tpr_min, _ = metrics.roc_curve(Mnist_y_test, y_pred_proba_min)
auc_min = np.round(metrics.roc_auc_score(Mnist_y_test, y_pred_proba_min), 3)
precision_min, recall_min, _ = metrics.precision_recall_curve(Mnist_y_test, y_pred_proba_min)
```

(2) logistic regression classifier with best value of the regularization parameter

In [ ]:

```
# fit model log regression with best reg
logistic_best = LogisticRegression(
    random_state=2018, penalty="l1", solver="liblinear", C=3793
).fit(Mnist_X_train, Mnist_y_train)

# define metrics
y_pred_proba_best = logistic_best.predict_proba(Mnist_X_test)[::, 1]
fpr_best, tpr_best, _ = metrics.roc_curve(Mnist_y_test, y_pred_proba_best)
auc_best = np.round(metrics.roc_auc_score(Mnist_y_test, y_pred_proba_best), 3)
precision_best, recall_best, _ = metrics.precision_recall_curve(
    Mnist_y_test, y_pred_proba_best
)
```

(3) LDA classifier

In [ ]:

```
# fit model with linear discriminant analysis
lda = LinearDiscriminantAnalysis().fit(Mnist_X_train, Mnist_y_train)

# define metrics
y_pred_proba_lda = lda.predict_proba(Mnist_X_test)[::, 1]
fpr_lda, tpr_lda, _ = metrics.roc_curve(Mnist_y_test, y_pred_proba_lda)
auc_lda = np.round(metrics.roc_auc_score(Mnist_y_test, y_pred_proba_lda), 3)
precision_lda, recall_lda, _ = metrics.precision_recall_curve(
    Mnist_y_test, y_pred_proba_lda
)
```

(4) RF classifier

In [ ]:

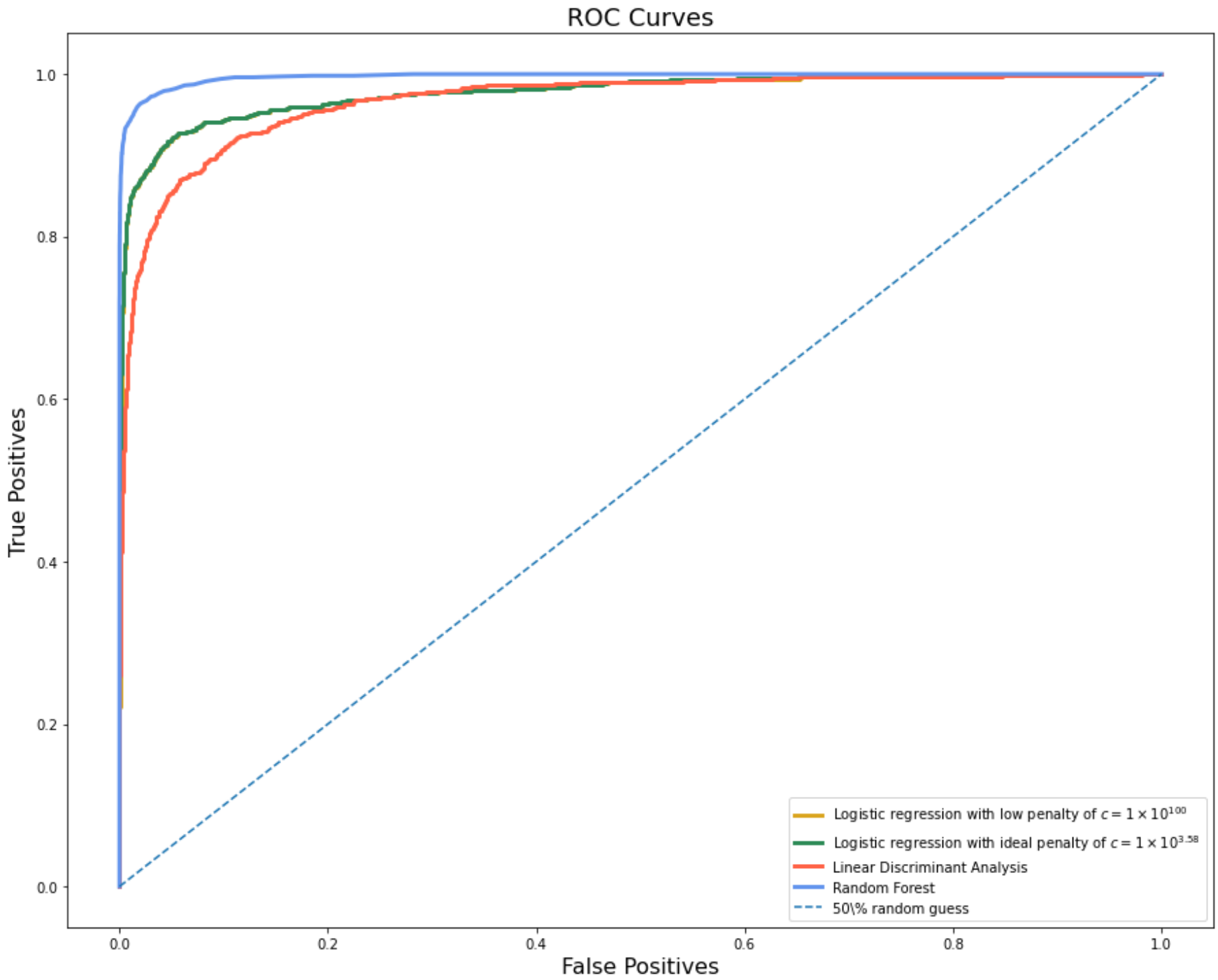
```
# fit model with rf
rf = RandomForestClassifier(random_state=2018).fit(Mnist_X_train, Mnist_y_train)

# define metrics
y_pred_proba_rf = rf.predict_proba(Mnist_X_test)[::, 1]
fpr_rf, tpr_rf, _ = metrics.roc_curve(Mnist_y_test, y_pred_proba_rf)
auc_rf = np.round(metrics.roc_auc_score(Mnist_y_test, y_pred_proba_rf), 3)
precision_rf, recall_rf, _ = metrics.precision_recall_curve(
```

```
    Mnist_y_test, y_pred_proba_rf
)
```

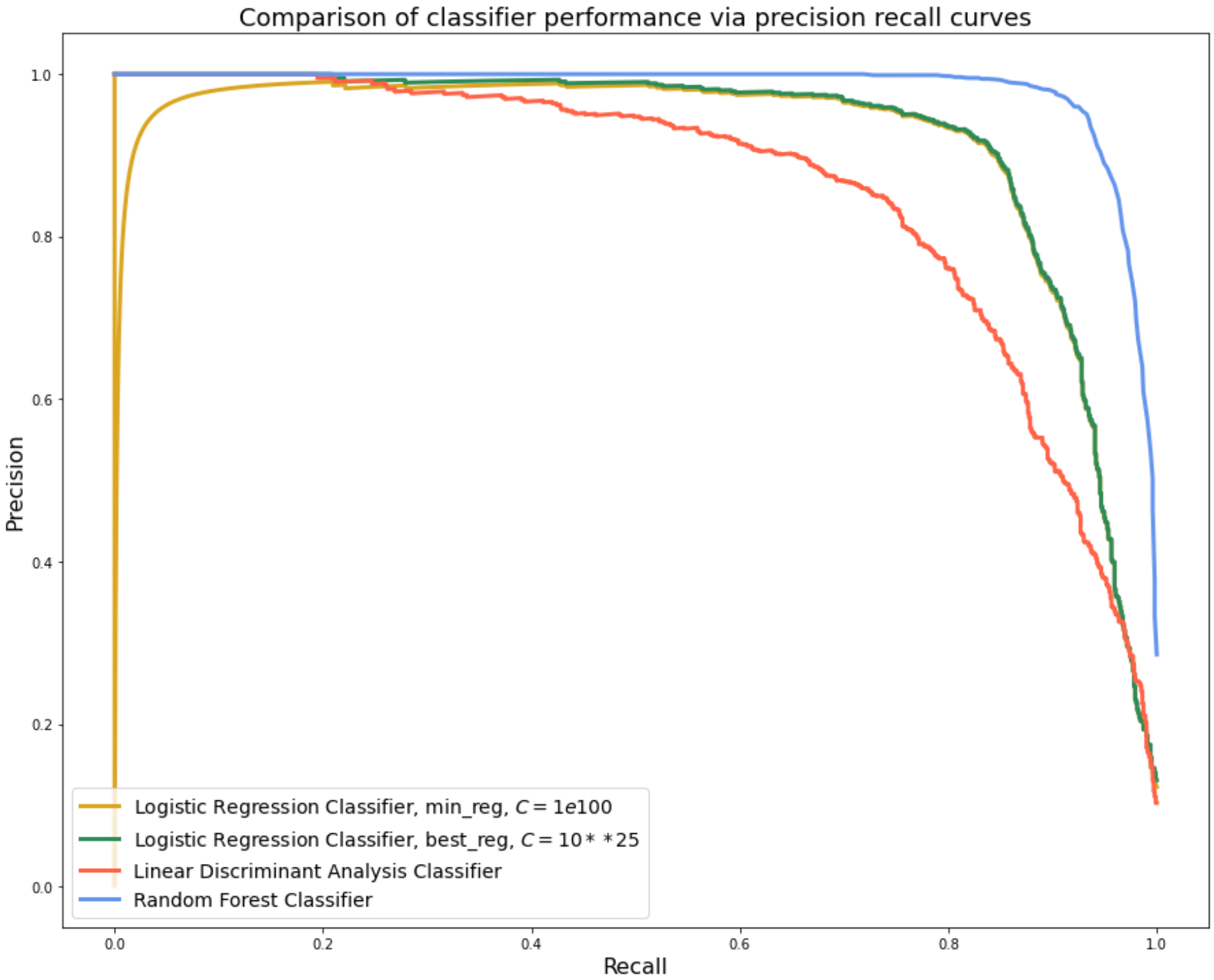
Compare classifier performance using ROC and PR curves

In [ ]:

```
fig, ax = plt.subplots(figsize = (15, 12))
ax.plot(fpr_min, tpr_min, label = "Logistic regression with low penalty of $c=1 \\times 10^{100}$", color = "goldenrod",
ax.plot(fpr_best, tpr_best, label = "Logistic regression with ideal penalty of $c=1 \\times 10^{3.58}$", color = "seagree
ax.plot(fpr_lda, tpr_lda, label = "Linear Discriminant Analysis", color = "tomato", linewidth = 3.0)
ax.plot(fpr_rf, tpr_rf, label = "Random Forest", color = "cornflowerblue", linewidth = 3.0)

ax.plot(np.arange(0, 1.1, 0.1), np.arange(0, 1.1, 0.1), "--", label="50\% random guess")
ax.legend(loc = 4, fontsize = 10)
ax.set_xlabel("False Positives", fontsize = 16)
ax.set_ylabel("True Positives", fontsize = 16)
ax.set_title("ROC Curves", fontsize = 18)
plt.show()
```


ROC Curves
True Positives
False Positives
0.0
0.2
0.4
0.6
0.8
1.0
Logistic regression with low penalty of $c = 1 \times 10^{100}$
Logistic regression with ideal penalty of $c = 1 \times 10^{3.58}$
Linear Discriminant Analysis
Random Forest
50\% random guess

In [ ]:
```
# make the PR curve
plt.figure(figsize = (15, 12))
plt.title("Comparison of classifier performance via precision recall curves", size = 18)

plt.plot(recall_min, precision_min, label = "Logistic Regression Classifier, min_reg, $C=1e100$", color = "goldenrod", li
plt.plot(recall_best, precision_best, label = "Logistic Regression Classifier, best_reg, $C=10 ** 25$", color = "seagreer
plt.plot(recall_lda, precision_lda, label = "Linear Discriminant Analysis Classifier", color = "tomato", linewidth = 3.0)
plt.plot(recall_rf, precision_rf, label = "Random Forest Classifier", color = "cornflowerblue", linewidth = 3.0)

plt.ylabel("Precision", size = 16)
plt.xlabel("Recall", size = 16)
plt.legend(loc = 3, prop = {'size': 14})
plt.show()
```


Comparison of classifier performance via precision recall curves
Precision
Recall
0.0
0.2
0.4
0.6
0.8
1.0
Logistic Regression Classifier, min_reg, C = 1e100
Logistic Regression Classifier, best_reg, C = 10**25
Linear Discriminant Analysis Classifier
Random Forest Classifier

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.

The best performance is seen with the random forest classifier when plotting both the ROC and the precision recall curves. The worst performance is seen with the linear discriminant analysis classifier. Both of the logistic regression models show almost exactly the same performance when plotting the ROC and precision recall curves, so the regularization does not seem to have made a big difference.