Assignment 3 - Supervised Learning: model training and evaluation

Yuanjing Zhu

Netid: yz792

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).

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 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* $\{(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \ldots, (y_N, \mathbf{x}_N)\}$.

(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.

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 [ ]:
```
# 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):
        # returns the value of the sigmoid
        pass

    # Cost function for an input set of weights
    def cost(self, X, y, w):
        # returns the average cross entropy cost
        pass

    # 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
        pass

    # 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, 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)

        # returns nothing
        pass

    # Use the trained model to predict the confidence scores
    def predict_proba(self, X):
        # returns the confidence score for the each sample
        pass

    # Use the trained model to make binary predictions
    def predict(self, X, thresh=0.5):
        # returns a binary prediction for each sample
        pass

    # 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 descent
        #  from the last model fitting process
        pass
```

```
    # Appends a column of ones as the first feature to account for the bias term
    def prepare_x(self, X):
        # returns the X with a new feature of all ones (a column that is the new column 0)
        pass
```

(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.

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?

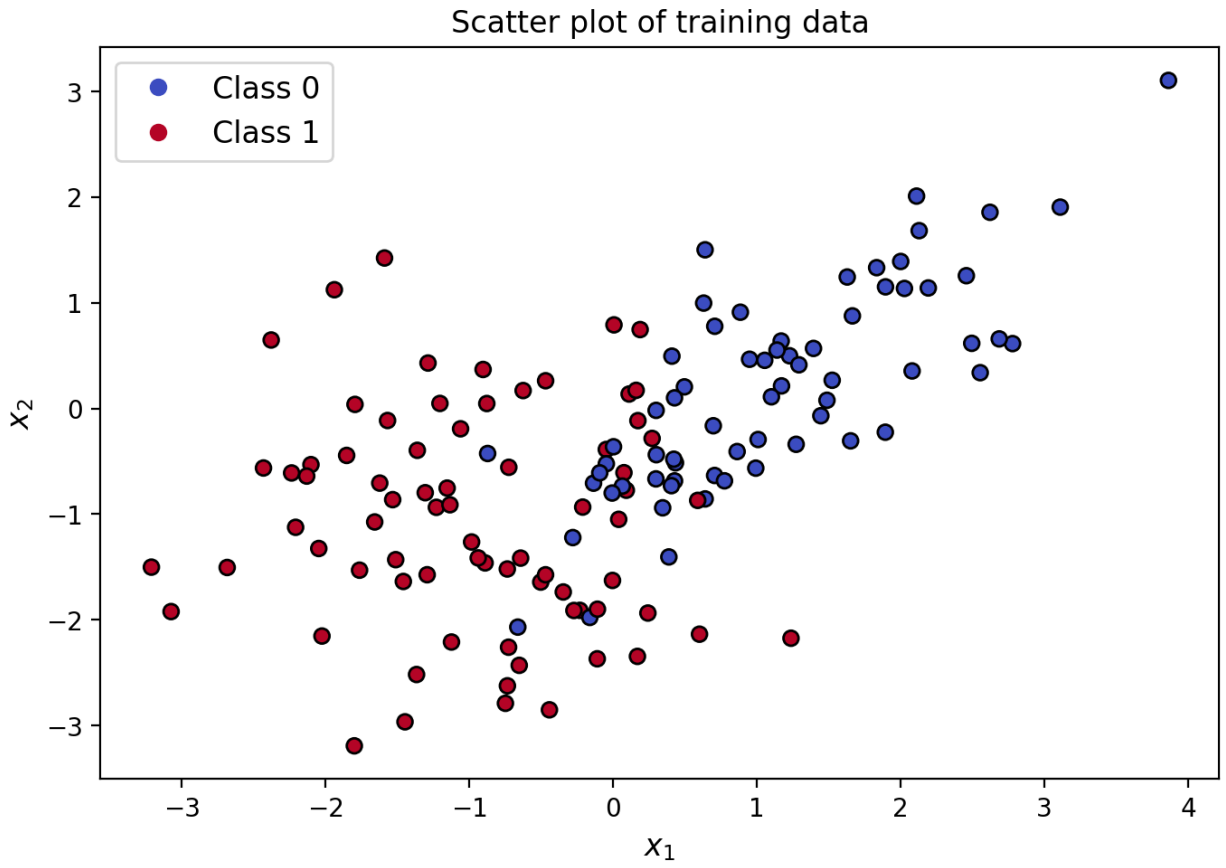
ANSWER

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import copy
import os
os.chdir('F:/Duke MIDS/705_ML/Assignment/03')

%config InlineBackend.figure_format = 'retina'
```

I. Load, prepare, and plot your data

(a) Load the data, train-test split, and plot the data.

In [ ]:
```
# Load the dataset
A3_Q1_data = pd.read_csv('A3_Q1_data.csv', header=0)
X = A3_Q1_data.iloc[:,0:2]
y = A3_Q1_data.iloc[:,2]

# 7:3 train-test split
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
# reset index
for i in [X_train, X_test, y_train, y_test]:
    i.index = range(i.shape[0])

# plot the training data by class
plt.figure(figsize=(7,5), dpi= 100)
scatter = plt.scatter(X_train.iloc[:, 0],
                      X_train.iloc[:, 1],
                      c=y_train,
                      edgecolors='black',
                      cmap=plt.cm.coolwarm)
plt.legend(handles=scatter.legend_elements()[0], labels=['Class 0', 'Class 1'], fontsize=12
plt.xlabel('$x_1$', fontsize=12)
plt.ylabel('$x_2$', fontsize=12)
plt.title("Scatter plot of training data")
plt.tight_layout()
plt.show()
```



The data appears to be **separable**, since it seems possible to draw a line that separates the two classes.

Logistic regression may be a good choice because 1) the outcome variable is binary, it's a classification problem, 2) the two classes seem to be linearly separable.

(b) Preprocessing the data to deal with missing values, scale differences, etc.

In [ ]:
```
# features shape
print('X_train shape: ', X_train.shape)
print('X_test shape: ', X_test.shape)
```

```
X_train shape:  (140, 2)
X_test shape:  (60, 2)
```

In [ ]:
```
# check missing values
print("Training set:")
for i in range(X_train.shape[1]):
    print(f"Feature {X_train.isna().sum().index[i]} has \
        {X_train.isna().sum().values[i]} missing values.")
print(f"Label y has {y_train.isna().sum()} missing values.")
print("====================================")

print("Test set:")
for i in range(X_test.shape[1]):
    print(f"Feature {X_test.isna().sum().index[i]} has \
        {X_test.isna().sum().values[i]} missing values.")
print(f"Label y has {y_test.isna().sum()} missing values.")
```

```
Training set:
Feature x1 has         0 missing values.
Feature x2 has         0 missing values.
Label y has 0 missing values.
====================================
Test set:
Feature x1 has         0 missing values.
Feature x2 has         0 missing values.
Label y has 0 missing values.
```

There are no missing values in the dataset.

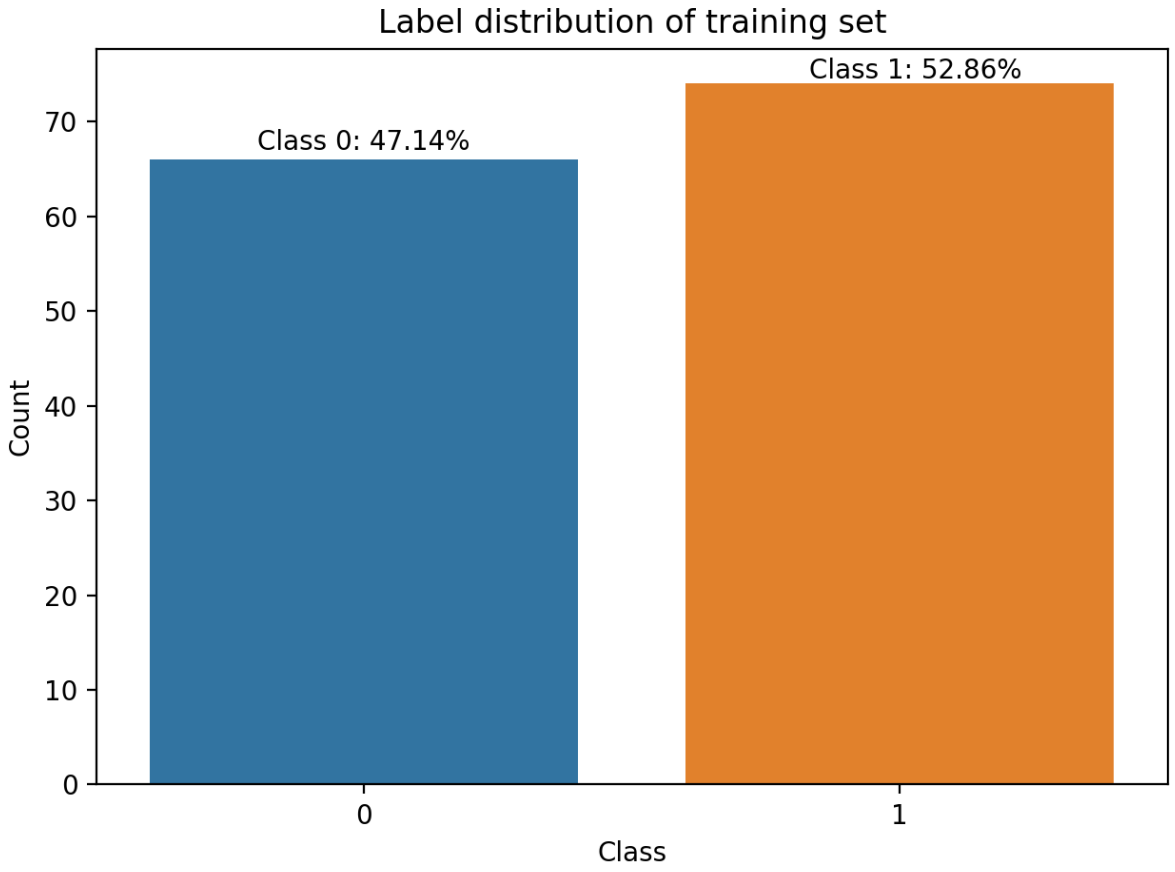
In [ ]:
```python
# check scale differences
for i in range(X_train.shape[1]):
    print(f"Feature: {X_train.columns[i]}")
    print(f"Mean: {(X_train.iloc[:,i].mean()):.2f}")
    print(f"Standard deviation: {(X_train.iloc[:,i].std()):.2f}")
    print("====================================")
```

```
Feature: x1
Mean: 0.01
Standard deviation: 1.40
====================================
Feature: x2
Mean: -0.48
Standard deviation: 1.18
====================================
```

In [ ]:
```python
# plot the histogram of training data
plt.Figure(figsize=(7,5), dpi= 100)
plt.hist(X_train.iloc[:,0], bins=20, alpha=0.5, label='$X_1$')
plt.hist(X_train.iloc[:,1], bins=20, alpha=0.5, label='$X_2$')
plt.xlabel('Feature value')
plt.ylabel('Frequency')
plt.legend(loc='upper right')
plt.title("Histogram of training data")
plt.show()
```

There is no scale difference between X_1 and X_2.

In [ ]:
```
# Label distribution
pd_y_train = y_train.to_frame()
percent0 = (len(pd_y_train[pd_y_train['y']==0])/pd_y_train.shape[0])*100
percent1 = (len(pd_y_train[pd_y_train['y']==1])/pd_y_train.shape[0])*100
sns.countplot(x = 'y', data = pd_y_train, dodge = False)
plt.xlabel("Class")
plt.ylabel("Count")
plt.title("Label distribution of training set")
plt.text(-0.2, 67, f"Class 0: {percent0:.2f}%")
plt.text(0.83, 74.5, f"Class 1: {percent1:.2f}%")
plt.tight_layout()
plt.show()
```



The data doesn't require any preprocessing because:

- There are no missing values.
- The scale of the data is already in the same range.
- The distribution of the two classes is balanced.

II. Stating the hypothesis set of models to evaluate ---- no output

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 the *entire training dataset* $\{(y_1, \mathbf{x}_1), (y_2, \mathbf{x}_2), \ldots, (y_N, \mathbf{x}_N)\}$.

The likelihood for one observation is :

$$L(\omega|y_i, x_i) = P(y_i = 1|x_i)^{y_i} P(y_i = 0|x_i)^{1-y_i}$$

The likelihood for all observations is:

$$\begin{aligned} L(\omega|y, x) &= P(y_1, y_2, \ldots, y_N|x_1, x_2, \ldots, x_N) \\ &= \prod_{i=1}^{N} P(y_i|x_i) \\ &= \prod_{i=1}^{N} P(y_i = 1|x_i)^{y_i} P(y_i = 0|x_i)^{1-y_i} \end{aligned}$$

For logistic regression:

$$P(y_i = 1|x_i) = \sigma(\omega^T x_i)$$

$$P(y_i = 0|x_i) = 1 - \sigma(\omega^T x_i)$$

Therefore, the likelihood for all observations can be written as:

$$L(\omega|y, x) = \prod_{i=1}^{N} \sigma(\omega^T x_i)^{y_i} (1 - \sigma(\omega^T x_i))^{1-y_i}$$

(d) 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.

Assuming $\widehat{y_i} = \sigma(\omega^T x_i)$:

$$\begin{aligned} L(\omega|y, x) &= \prod_{i=1}^{N} \sigma(\omega^T x_i)^{y_i} (1 - \sigma(\omega^T x_i))^{1-y_i} \\ &= \prod_{i=1}^{N} \widehat{y_i}^{y_i} (1 - \widehat{y_i})^{1-y_i} \end{aligned}$$

Take the log of both sides:

$$\begin{aligned} log(L(\omega|y, X)) &= log(\prod_{i=1}^{N} \widehat{y_i}^{y_i} (1 - \widehat{y_i})^{1-y_i}) \\ &= \sum_{i=1}^{N} log(\widehat{y_i}^{y_i} (1 - \widehat{y_i})^{1-y_i}) \\ &= \sum_{i=1}^{N} y_i log(\widehat{y_i}) + (1 - y_i) log(1 - \widehat{y_i}) \end{aligned}$$

Cost function:

$$\begin{aligned} C(\omega) &= -\frac{1}{N} log(L(\omega|y, X)) \\ &= -\frac{1}{N} \left[\sum_{i=1}^{N} y_i log(\widehat{y_i}) + (1 - y_i) log(1 - \widehat{y_i}) \right] \\ &= -\frac{1}{N} \left[\sum_{i=1}^{N} y_i log(\sigma(\omega^T x_i)) + (1 - y_i) log(1 - \sigma(\omega^T x_i)) \right] \end{aligned}$$

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

$$
\begin{aligned}
C(\omega) &= -\frac{1}{N}\left[\sum_{i=1}^{N} y_i log(\widehat{y}_i) + (1-y_i) log(1-\widehat{y}_i)\right] \\
\widehat{y}_i &= \sigma(\omega^T x_i) \\
&= \sigma(z_i) \\
z_i &= \omega^T x_i \\
&= \omega_0 x_{i0} + \omega_1 x_{i1} + \ldots + \omega_j x_{ij} \\
x_{ij} &: \text{feature j of sample i}
\end{aligned}
$$

According to the chain rule:

$$
\begin{aligned}
\frac{\partial C(\omega)}{\partial \omega_j} &= -\frac{1}{N}\sum_{i=1}^{N}\left(\frac{\partial C(\omega)}{\partial \widehat{y}_i} \cdot \frac{\partial \widehat{y}_i}{\partial z_i} \cdot \frac{\partial z_i}{\partial \omega_j}\right) \\
&= -\frac{1}{N}\sum_{i=1}^{N}\left(\left(-\frac{y_i}{\widehat{y}_i} + \frac{1-y_i}{1-\widehat{y}_i}\right) \cdot \widehat{y}_i(1-\widehat{y}_i) \cdot x_{ij}\right) \\
&= \frac{1}{N}\sum_{i=1}^{N}\left(\widehat{y}_i - y_i\right) x_{ij} \\
&= \frac{1}{N}\sum_{i=1}^{N}\left(\sigma(\omega^T x_i) - y_i\right) x_{ij}
\end{aligned}
$$

With respect to each of the parameters:

$$
\begin{aligned}
\nabla_{\mathbf{w}} C(\mathbf{w}) &= \left[\frac{\partial C}{\partial w_0}, \frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}\right] \\
&= \begin{bmatrix} \frac{1}{N}\sum_{i=1}^{N}\left(\sigma(\omega^T x_i) - y_i\right) x_{i0} \\ \frac{1}{N}\sum_{i=1}^{N}\left(\sigma(\omega^T x_i) - y_i\right) x_{i1} \\ \frac{1}{N}\sum_{i=1}^{N}\left(\sigma(\omega^T x_i) - y_i\right) x_{i2} \end{bmatrix}^T
\end{aligned}
$$

(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.

$$
\begin{aligned}
w_j^{(k+1)} &= w_j^{(k)} - \eta * \nabla_{\mathbf{w}} C(\mathbf{w}) \\
&= w_j^{(k)} - \frac{\eta}{N}\sum_{i=1}^{N}\left(\sigma(\omega^T x_i) - y_i\right) x_{ij}
\end{aligned}
$$

With respect to each of the parameters:

$$\omega^{(k+1)} = \begin{bmatrix} \omega_0^{(k+1)} \\ \omega_1^{(k+1)} \\ \omega_2^{(k+1)} \end{bmatrix}^T$$
$$= \begin{bmatrix} \omega_0^{(k)} - \frac{\eta}{N} \sum_{i=1}^{N} \left(\sigma(\omega^T x_i) - y_i\right) x_{i0} \\ \omega_1^{(k)} - \frac{\eta}{N} \sum_{i=1}^{N} \left(\sigma(\omega^T x_i) - y_i\right) x_{i1} \\ \omega_2^{(k)} - \frac{\eta}{N} \sum_{i=1}^{N} \left(\sigma(\omega^T x_i) - y_i\right) x_{i2} \end{bmatrix}^T$$

IV. Implement gradient descent and your logistic regression algorithm

(g) Implement your logistic regression model.

In [ ]:
```python
# Logistic regression class
class Logistic_regression:
    # Class constructor
    def __init__(self):
        self.w = None     # logistic regression weights
        self.saved_w = [] # save weights at each iteration
        # returns nothing
        pass

    # Appends a column of ones as the first feature to account for the bias term
    def prepare_x(self, X):
        '''
        Add a column of ones to the input matrix X to account for the bias term
        Input: X - a numpy array of shape (n, m)
        Output: a numpy array of shape (n, m+1)
        '''
        return np.hstack((np.ones((X.shape[0], 1)), X))

    # Method for calculating the sigmoid function of w^T X for an input set of weights
    def sigmoid(self, X, w):
        '''
        Calculate the sigmoid function of w^T X, X is transformed.
        Input: X - a numpy array of shape (n, m+1)
               w - a numpy array of shape (m+1, 1)
        Output: the value of the sigmoid
        '''
        return 1/(1+np.exp(-np.dot(X, w)))

    # Cost function for an input set of weights
    def cost(self, X, y, w):
        '''
        Calculate the average cross entropy cost
        Input: X - a numpy array of shape (n, m+1)
               y - a numpy array of shape (n, 1)
               w - a numpy array of shape (m+1, 1)
        Output: the average cross entropy cost
        '''
        y_hat = self.sigmoid(X=X, w=w)
        avg_cost = -np.mean(y * np.log(y_hat) + (1-y) * np.log(1-y_hat))
        return avg_cost
```

```
    # Update the weights in an iteration of gradient descent
    def gradient_descent(self, X, y, lr):
        '''
        Update the weights in an iteration of gradient descent
        Input: X - a numpy array of shape (n, m+1)
               y - a numpy array of shape (n, 1)
               lr - the learning rate
        Output: the magnitude of the Euclidean norm of the change in the weights'''
        n_samples = X.shape[0]
        y_hat = self.sigmoid(X=X, w=self.w)
        dw = (np.dot(X.T, (y_hat - y)) / n_samples)
        self.w -= dw * lr        # update the weights
        delta_w = np.linalg.norm(dw * lr)  # calculate the magnitude of the
                                           # Euclidean norm of the change in the weights
        return delta_w


    # Fit the logistic regression model to the data through gradient descent
    def fit(self, X, y, lr, delta_thresh=1e-6, max_iter=5000, verbose=False):
        '''
        Fit the logistic regression model to the data through gradient descent
        Input: X - a numpy array of shape (n, m+1)
               y - a numpy array of shape (n, 1)
               lr - the learning rate
               delta_thresh - the threshold for the magnitude of the Euclidean norm
                            of the change in the weights to stop the gradient descent
               max_iter - maximum number of iterations of gradient descent, default 5000
               verbose - a boolean flag to print out the weights at each iteration
        Output: None
        '''
        # 1. prepare X, initialize weights
        np.random.seed(42) # set the random seed for reproducibility
        # initialize weights to a different values between 0 and 1
        w_init = np.random.rand(X.shape[1])
        self.w = w_init
        self.saved_w.append(copy.deepcopy(self.w))

        # 2. start iteration
        for i in range(max_iter):
            delta_w = self.gradient_descent(X=X, y=y, lr=lr)
            self.saved_w.append(copy.deepcopy(self.w))
            if delta_w < delta_thresh:
                break
            if verbose == True:
                print(f"Iter {i}: weights {self.w}")
        # returns nothing

    # Use the trained model to predict the confidence scores
    def predict_proba(self, X):
        '''
        Predict the confidence scores
        Input: X - a numpy array of shape (n, m+1) after transformation
        Output: a numpy array of shape (n, 1)
        '''
        pred_prob = self.sigmoid(X=X, w=self.w)
        return pred_prob

    # Use the trained model to make binary predictions
    def predict(self, X, thresh=0.5):
        '''
```

```
        Predict the binary class labels
        Input: X - a numpy array of shape (n, m+1)
               thresh - the threshold for the probability of each class
        Output: a binary prediction for each sample
        '''
        pred_prob = self.predict_proba(X=X)
        y_pred_class = [1 if i > thresh else 0 for i in pred_prob]
        return y_pred_class


    # Stores the learning curves from saved weights from gradient descent
    def learning_curve(self, X, y):
        '''
        Stores the learning curves from saved weights from gradient descent
        Input: X - a numpy array of shape (n, m+1)
               y - a numpy array of shape (n, 1)
        Output: a list of the cost function values from each iteration of gradient descent
        '''
        # returns the value of the cost function from each step in gradient descent
        #  from the last model fitting process
        cost_l = []
        for i in self.saved_w:
            cost_l.append(self.cost(X=X, y=y, w=i))
        return cost_l
```

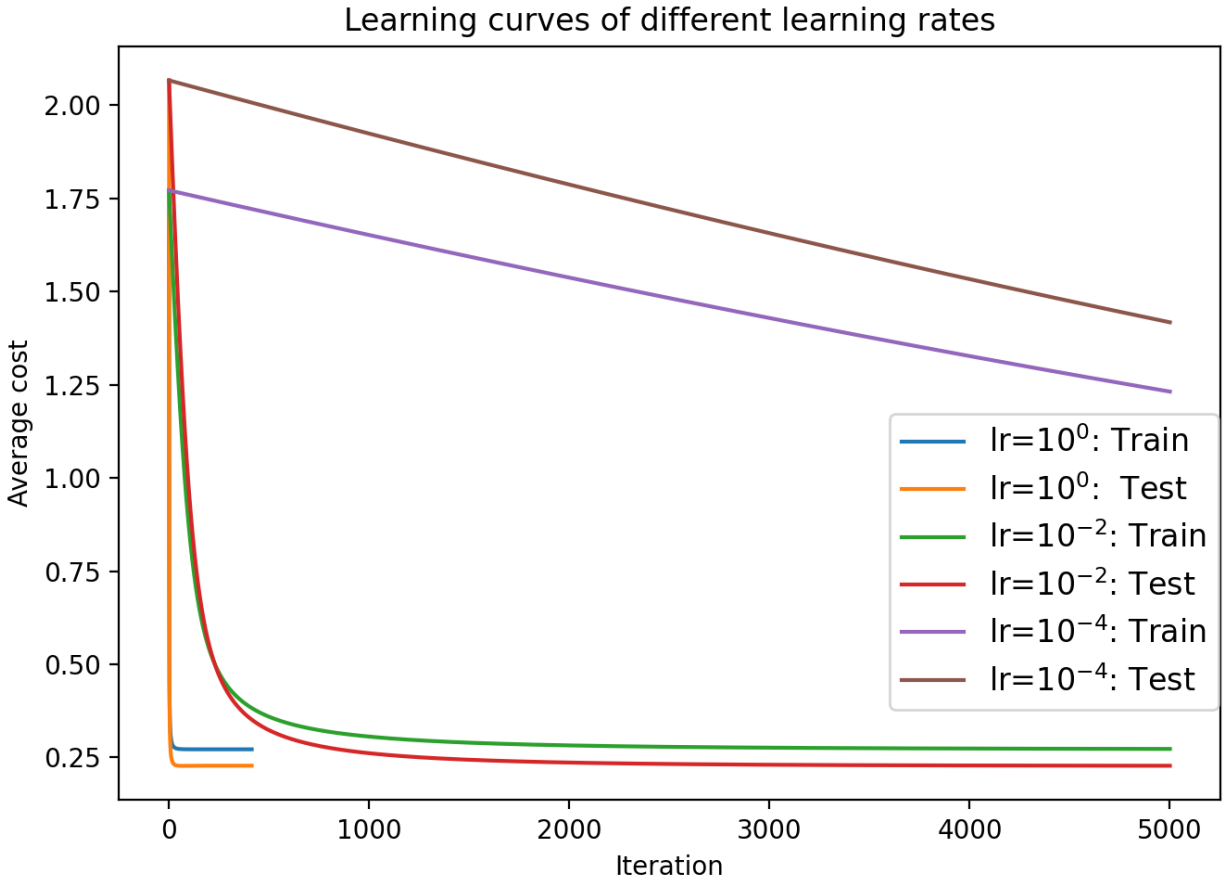
(h) Choose a learning rate from $\{10^{-0}, 10^{-2}, 10^{-4}\}$ to fit the model. Plot both the training and test data average cost as a function of each iteration of gradient descent.

In [ ]:
```
# prepare the data
X = X.values
y = y.values
X_train = X_train.values
y_train = y_train.values
X_test = X_test.values
y_test = y_test.values
```

In [ ]:
```
# initialize logistic regression model
my_lr = Logistic_regression()

# add a column of ones to X_train and X_test
X_train_new = my_lr.prepare_x(X_train)
X_test_new = my_lr.prepare_x(X_test)

# iterate through 3 different learning rates, fit the model, and store the learning curves
l_lr = [10**(0), 10**(-2), 10**(-4)]
avg_cost_tr = []
avg_cost_te = []
for i in range(len(l_lr)):
    my_lr = Logistic_regression()
    my_lr.fit(X=X_train_new, y=y_train, lr=l_lr[i], verbose=False)
    avg_cost_tr.append(my_lr.learning_curve(X=X_train_new, y=y_train))
    avg_cost_te.append(my_lr.learning_curve(X=X_test_new, y=y_test))
```

In [ ]:
```
# plot the learning curves
plt.figure(figsize=(7,5), dpi= 100)
plt.plot(avg_cost_tr[0], label='lr=$10^{0}$: Train')
plt.plot(avg_cost_te[0], label='lr=$10^{0}$:  Test')
plt.plot(avg_cost_tr[1], label='lr=$10^{-2}$: Train')
plt.plot(avg_cost_te[1], label='lr=$10^{-2}$: Test')
```

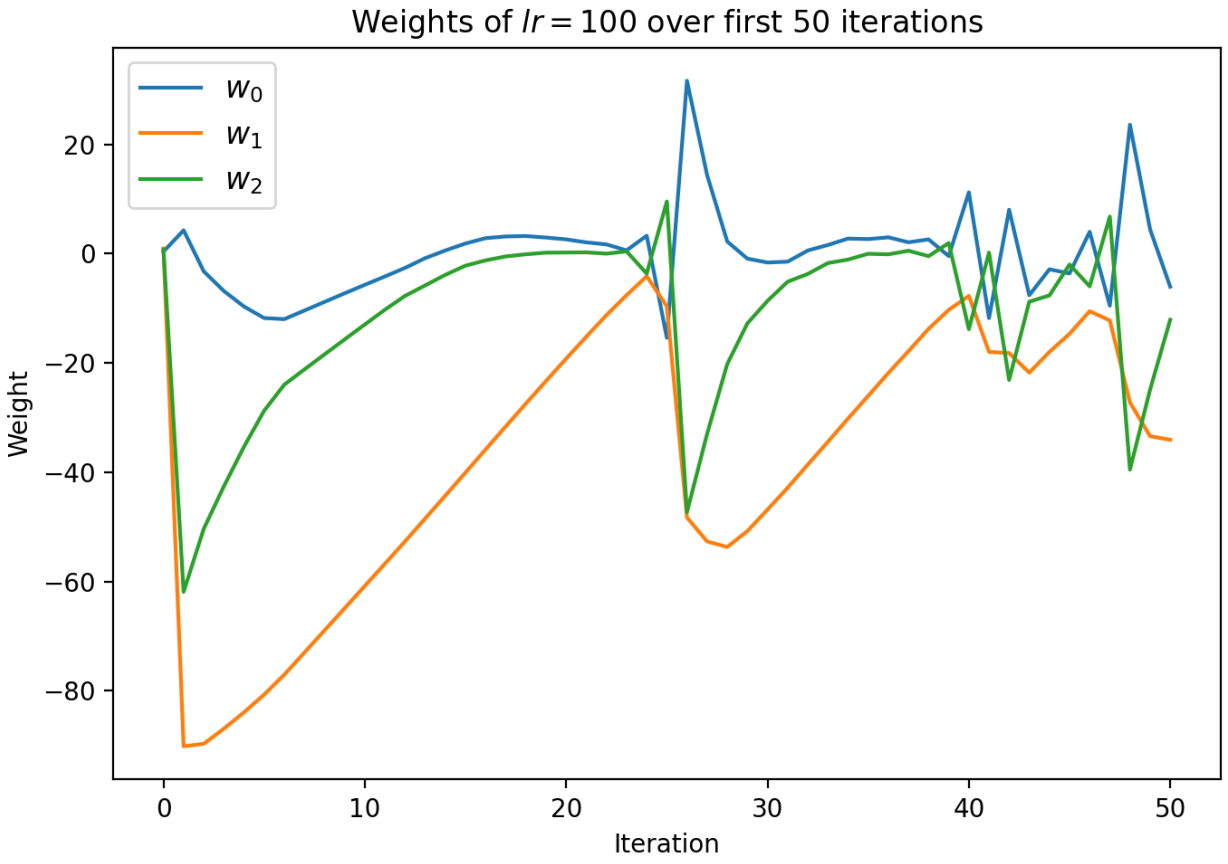
```
plt.plot(avg_cost_tr[2], label='lr=$10^{-4}$: Train')
plt.plot(avg_cost_te[2], label='lr=$10^{-4}$: Test')
plt.legend(loc = (0.7, 0.12), fontsize = 12)
plt.xlabel('Iteration')
plt.ylabel('Average cost')
plt.title('Learning curves of different learning rates')
plt.tight_layout()
plt.show()
```

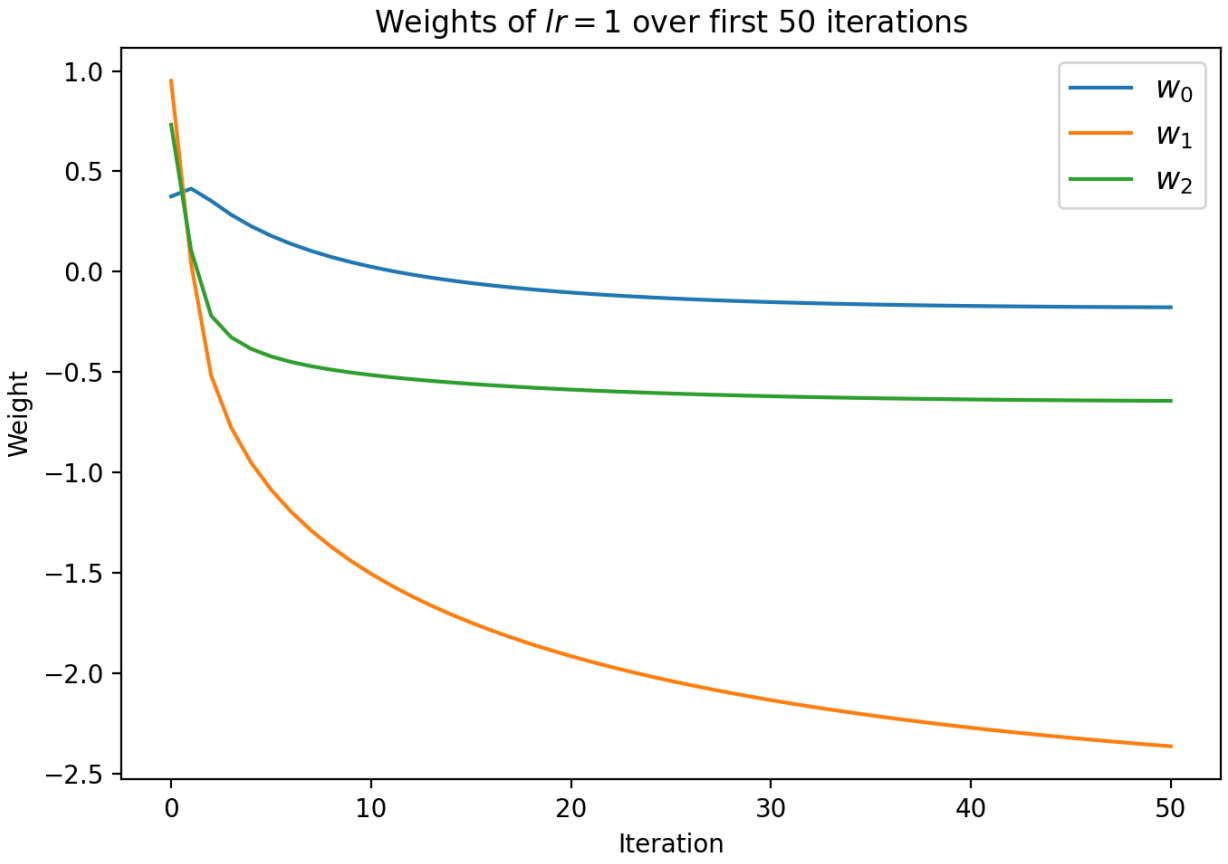


- Try running this process for a really big learning rate for this problem: 10^2. Look at the weights that the fitting process generates over the first 50 iterations and how they change.

In [ ]:
```
# lr = 100
my_lr = Logistic_regression()
my_lr.fit(X=X_train_new, y=y_train, lr=10**2)
weights_lr100 = my_lr.saved_w
# get each weight from the list of weights
w0_lr100 = [i[0] for i in weights_lr100]
w1_lr100 = [i[1] for i in weights_lr100]
w2_lr100 = [i[2] for i in weights_lr100]

# lr = 1
my_lr = Logistic_regression()
my_lr.fit(X=X_train_new, y=y_train, lr=1)
weights_lr1 = my_lr.saved_w
# get each weight from the list of weights
w0_lr1 = [i[0] for i in weights_lr1]
w1_lr1 = [i[1] for i in weights_lr1]
w2_lr1 = [i[2] for i in weights_lr1]
```

In [ ]:
```
# plot weights over first 50 iterations
plt.figure(figsize=(7,5), dpi= 100)
plt.plot(w0_lr100[:51], label='$w_0$')
plt.plot(w1_lr100[:51], label='$w_1$')
plt.plot(w2_lr100[:51], label='$w_2$')
plt.xlabel('Iteration')
plt.ylabel('Weight')
plt.title('Weights of $lr=100$ over first 50 iterations')
plt.legend(fontsize = 12)
plt.tight_layout()
plt.show()

plt.figure(figsize=(7,5), dpi= 100)
plt.plot(w0_lr1[:51], label='$w_0$')
plt.plot(w1_lr1[:51], label='$w_1$')
plt.plot(w2_lr1[:51], label='$w_2$')
plt.xlabel('Iteration')
plt.ylabel('Weight')
plt.title('Weights of $lr=1$ over first 50 iterations')
plt.legend(fontsize = 12)
plt.tight_layout()
plt.show()
```





When $lr = 100$, weights oscillate rapidly in a large range and it doesn't seem to converge to a solution.
On the other hand, when $lr = 1$, weights are updated more slowly in each iteration, and the updates of the weights are smoother and less oscillatory, leading to convergence after 50 iterations.

- What is the impact that the different values of learning have on the speed of the process and the results?

1. A very high learning rate (e.g. $lr = 100$) will lead to oscillatory updates of the weights, which may not converge to a solution.
2. A relatively high learning rate (e.g. $lr = 1$) will lead to rapid convergence, but it may not converge to the global minimum and thus lead to poor prediction results.
3. A very low learning rate (e.g. $lr = 10^-4$) will lead to slow convergence, and it may take a long time to get a good solution, which can be computationally expensive. Additionally, a low learning rate may be unable to escape from a local minimum, which can result in suboptimal results.
4. Overall, a moderate learning rate (e.g. $lr = 0.01$) strikes a good balance between the speed of convergence and the quality of the solution. Finding a suitable learning rate is a key step in the tuning the hyperparameters of a model.

- Of the options explored, what learning rate do you prefer and why?
- Use your chosen learning rate for the remainder of this problem.

In [ ]:
```
from sklearn.metrics import accuracy_score
```

```
# define a funtion to calculate the accuracy score for each learning rate
def eval_lr(lr):
    '''
    Evaluate the accuracy score for each learning rate
    Input: lr - the learning rate
    Output: the accuracy score for each learning rate
    '''
    my_lr = Logistic_regression()
    my_lr.fit(X=X_train_new, y=y_train, lr=lr)
    y_pred_train = my_lr.predict(X=X_train_new)
    y_pred_test = my_lr.predict(X=X_test_new)
    print(f"lr: {lr}")
    print(f"Accuracy on test set: {accuracy_score(y_test, y_pred_test):.2f}")
    print('====================================')

# evaluate the accuracy score for each learning rate
eval_lr(10**2)
eval_lr(10**0)
eval_lr(10**(-2))
eval_lr(10**(-4))
```

```
lr: 100
Accuracy on test set: 0.85
====================================
lr: 1
Accuracy on test set: 0.90
====================================
lr: 0.01
Accuracy on test set: 0.90
====================================
lr: 0.0001
Accuracy on test set: 0.10
====================================
```

I choose $lr = 10^{-2}$. I rule out $lr = 10^{-4}$ because it is too slow to converge and doesn't converge after 5000 iterations. I rule out $lr = 1$ because the learning rate is too large and it doesn't converge to the global minimum, leading to low accuracy on test set. $lr = 1$ and $lr = 0.01$ both converge to the global minimum and the accuracy of test data is similar. However, according to the plot of the average cost, $lr = 0.01$ strikes a better balance between the speed of convergence and stability of convergence. Therefore, I choose $lr = 0.01$ for further analysis.

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.

In [ ]:
```
from sklearn.neighbors import KNeighborsClassifier as KNC
from sklearn.model_selection import StratifiedKFold
from sklearn.inspection import DecisionBoundaryDisplay
```

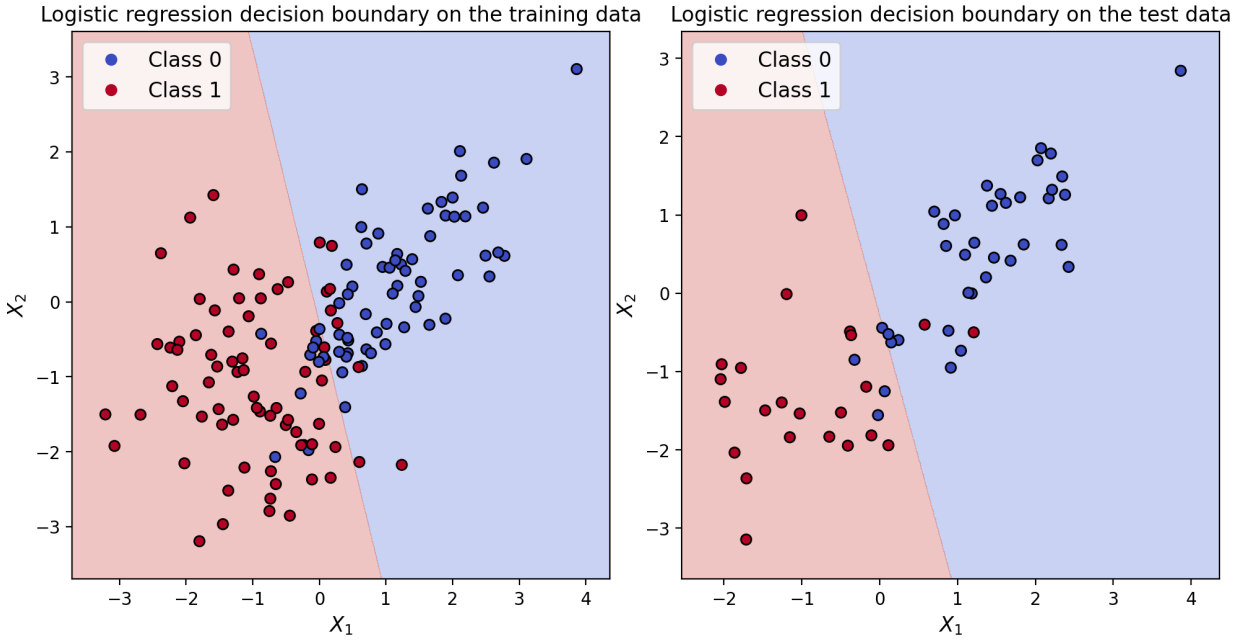
In [ ]:
```
# fit logistic regression model with learning rate of 0.01
my_lr = Logistic_regression()
```

```
my_lr.fit(X=X_train_new, y=y_train, lr=0.01)

# KNN model
knn = KNC(n_neighbors=7).fit(X_train, y_train)
```

- 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.

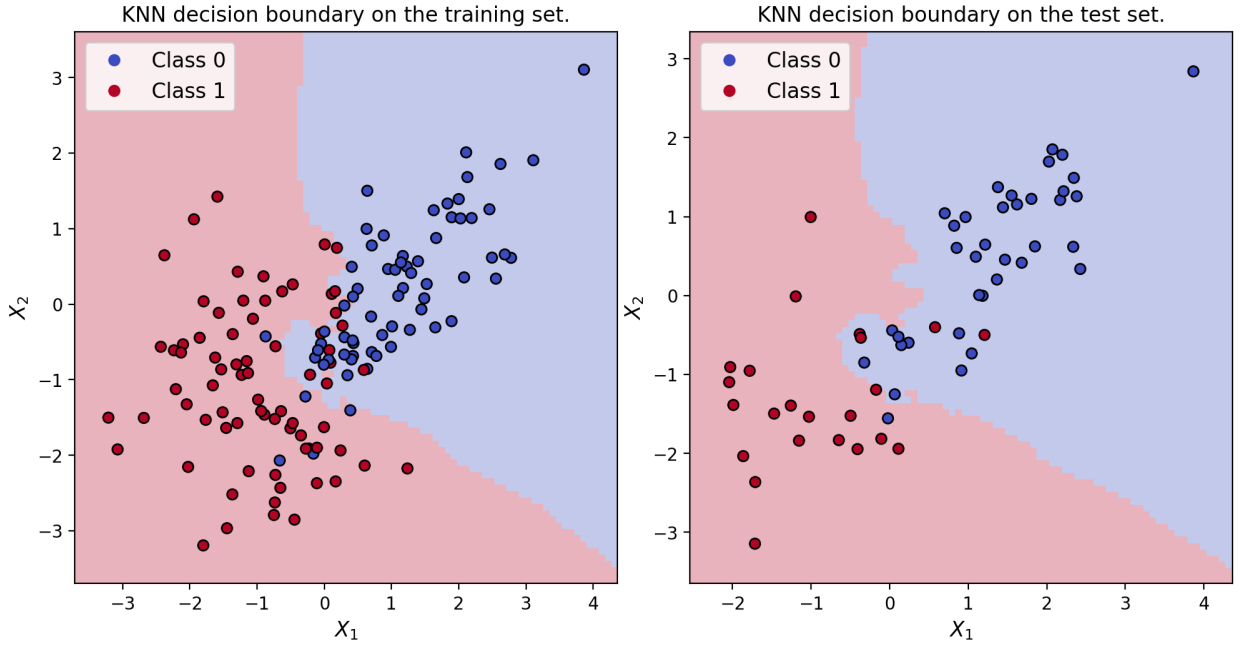
In [ ]:
```
# plot the decision boundary of logistic regression model
# Prepare for plotting training data
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1

xx_train, yy_train = np.meshgrid(np.arange(x_min, x_max, 0.01),
                                 np.arange(y_min, y_max, 0.01))
xx_yy = np.c_[xx_train.ravel(), yy_train.ravel()]
xx_yy_ = my_lr.prepare_x(xx_yy)
Z_train = my_lr.predict(xx_yy_)
Z_train = np.array(Z_train).reshape(xx_train.shape)

# Prepare for plotting test data
x_min, x_max = X_test[:, 0].min() - 1, X_test[:, 0].max() + 1
y_min, y_max = X_test[:, 1].min() - 1, X_test[:, 1].max() + 1

xx_test, yy_test = np.meshgrid(np.arange(x_min, x_max, 0.01),
                               np.arange(y_min, y_max, 0.01))
xx_yy = np.c_[xx_test.ravel(), yy_test.ravel()]
xx_yy_ = my_lr.prepare_x(xx_yy)
Z_test = my_lr.predict(xx_yy_)
Z_test = np.array(Z_test).reshape(xx_test.shape)

# wrap into lists
list_X = [X_train, X_test]
list_y = [y_train, y_test]
list_xx = [xx_train, xx_test]
list_yy = [yy_train, yy_test]
list_Z = [Z_train, Z_test]
list_title = ['training data', 'test data']

# Plot the decision boundary of training and test data
fig, ax = plt.subplots(1, 2, figsize=(10, 5), dpi= 100)
fig.tight_layout()
for i in range(2):
    # plot the decision boundary
    ax[i].contourf(list_xx[i], list_yy[i], list_Z[i],
                   cmap=plt.cm.coolwarm, alpha=0.3)
    # plot the data
    scatter = ax[i].scatter(list_X[i][:, 0], list_X[i][:, 1],
                            c=list_y[i], cmap=plt.cm.coolwarm, edgecolor='black')
    # set plot attributes
    ax[i].legend(handles=scatter.legend_elements()[0],
                 labels=['Class 0', 'Class 1'], fontsize = 12)
    ax[i].set_xlabel('$X_1$', fontsize = 12)
    ax[i].set_ylabel('$X_2$', fontsize = 12)
    ax[i].set_xlim(list_X[i][:, 0].min()-0.5, list_X[i][:, 0].max()+0.5)
    ax[i].set_ylim(list_X[i][:, 1].min()-0.5, list_X[i][:, 1].max()+0.5)
```

```
    ax[i].set_title(f'Logistic regression decision boundary on the {list_title[i]}')
plt.show()
```



In [ ]:
```
# plot the decision boundary of the KNN model
# wrap into lists
list_X = [X_train, X_test]
list_y = [y_train, y_test]
list_title = ['training set', 'test set']

# start plotting
fig, ax = plt.subplots(1, 2)
fig.set_size_inches(10, 5)
fig.tight_layout()
for i in range(2):
    # plot the decision boundary
    DecisionBoundaryDisplay.from_estimator(
        knn,
        list_X[i],
        ax=ax[i],
        response_method='predict',
        plot_method='pcolormesh',
        cmap = plt.cm.coolwarm,
        shading='auto',
        alpha=0.3)
    # plot the data points
    scatter = ax[i].scatter(list_X[i][:, 0],
                            list_X[i][:, 1],
                            c=list_y[i],
                            cmap=plt.cm.coolwarm,
                            edgecolors='black')
    # set the title, legend, and axis labels
    ax[i].legend(handles=scatter.legend_elements()[0],
                 labels=['Class 0', 'Class 1'], fontsize = 12,
                 loc='upper left')
    ax[i].set_xlabel('$X_1$', fontsize = 12)
    ax[i].set_ylabel('$X_2$', fontsize = 12)
    ax[i].set_title(f"KNN decision boundary on the {list_title[i]}.")
    ax[i].set_xlim(list_X[i][:, 0].min() - 0.5, list_X[i][:, 0].max() + 0.5)
```

```
    ax[i].set_ylim(list_X[i][:, 1].min() - 0.5, list_X[i][:, 1].max() + 0.5)
plt.show()
```



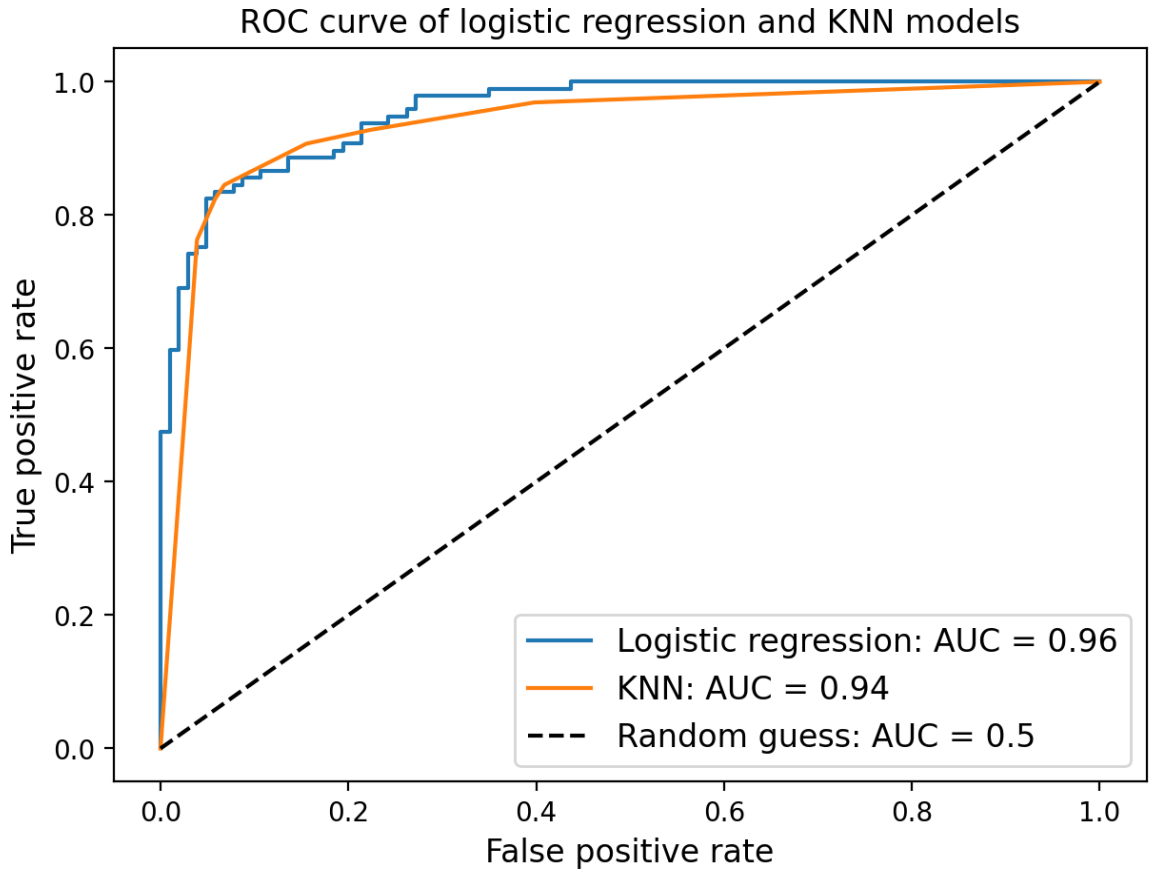
Source for plotting logistic regression decision boundary: https://favtutor.com/blogs/decision-boundary-logistic-regression

- 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.

In [ ]:
```
from sklearn.metrics import roc_auc_score, roc_curve

# define the cross validation method
cv = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)

# loop through different training and test sets to fit the model
y_pred_prob_lr = []
y_pred_prob_knn = []
y_true = []
for train_index, test_index in cv.split(X, y):
    X_train_cv, X_test_cv = X[train_index, :], X[test_index, :]
    y_train_cv, y_test_cv = y[train_index], y[test_index]
    y_true.extend(y_test_cv)   # store the true labels

    # fit logistic regression model
    X_train_cv_ = my_lr.prepare_x(X_train_cv)
    X_test_cv_ = my_lr.prepare_x(X_test_cv)
    lr_cv = Logistic_regression()
    lr_cv.fit(X_train_cv_, y_train_cv, lr=0.01)
```

```
    y_pred_prob_lr.extend(lr_cv.predict_proba(X_test_cv_))

    # fit KNN model
    knn_cv = KNC(n_neighbors=7).fit(X_train_cv, y_train_cv)
    y_pred_prob_knn.extend(knn_cv.predict_proba(X_test_cv)[:, 1])
```

In [ ]:
```
fpr_lr, tpr_lr, _ = roc_curve(y_true, y_pred_prob_lr)
fpr_knn, tpr_knn, _ = roc_curve(y_true, y_pred_prob_knn)
auc_lr = roc_auc_score(y_true, y_pred_prob_lr)
auc_knn = roc_auc_score(y_true, y_pred_prob_knn)

# plot the ROC curve
plt.figure(figsize=(7, 5), dpi=100)
plt.plot(fpr_lr, tpr_lr, label=f'Logistic regression: AUC = {auc_lr:.2f}')
plt.plot(fpr_knn, tpr_knn, label=f'KNN: AUC = {auc_knn:.2f}')
plt.plot([0, 1], [0, 1], 'k--', label='Random guess: AUC = 0.5')
plt.xlabel('False positive rate', fontsize=12)
plt.ylabel('True positive rate', fontsize=12)
plt.title('ROC curve of logistic regression and KNN models')
plt.legend(fontsize=12)
plt.show()
```



- 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?

The sample size is small in the case. The training set only has 140 samples, and the test set only has 60

samples. Cross validation helps to build a model that is more robust and less susceptible to random split of training and test sets. Cross validation also helps prevent overfitting when sample size is small.

In terms of ROC curves, logistic regression outperforms KNN since the auc score is higher. In terms of decision boundaries, KNN separates the trainning set better than logistic regression, but logistic regression separates the test set better than KNN, which indicates overfitting of KNN. Therefore, I would select **logistic regression** to use on previously unseen data for this problem.

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.

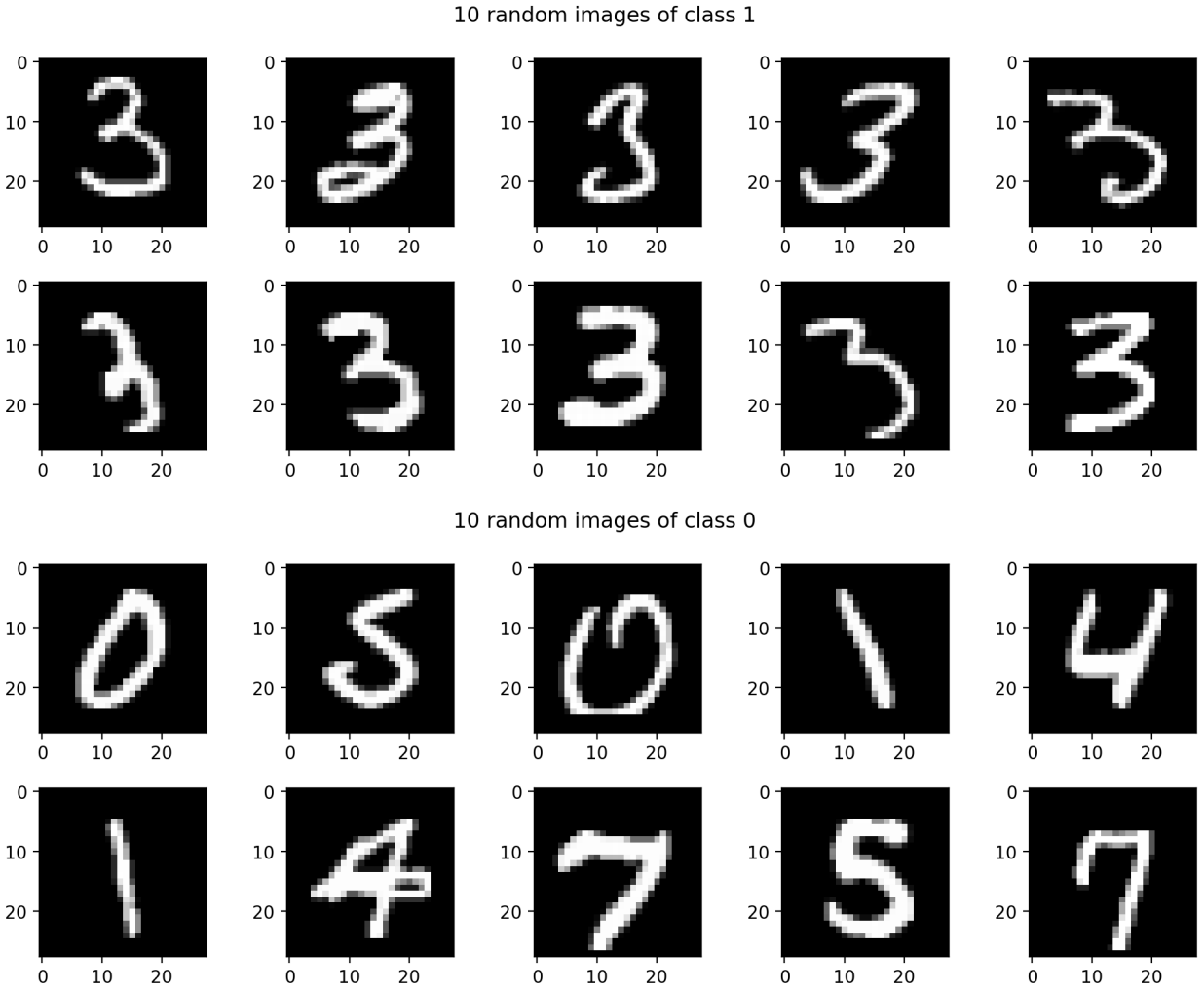
In [ ]:
```
# Load the MNIST Data
from sklearn.datasets import fetch_openml
from sklearn.model_selection import train_test_split
import numpy as np
import matplotlib.pyplot as plt
import pickle

# Set this to True to download the data for the first time and False after 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 into a training and test split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=1/7, random_state=8

    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

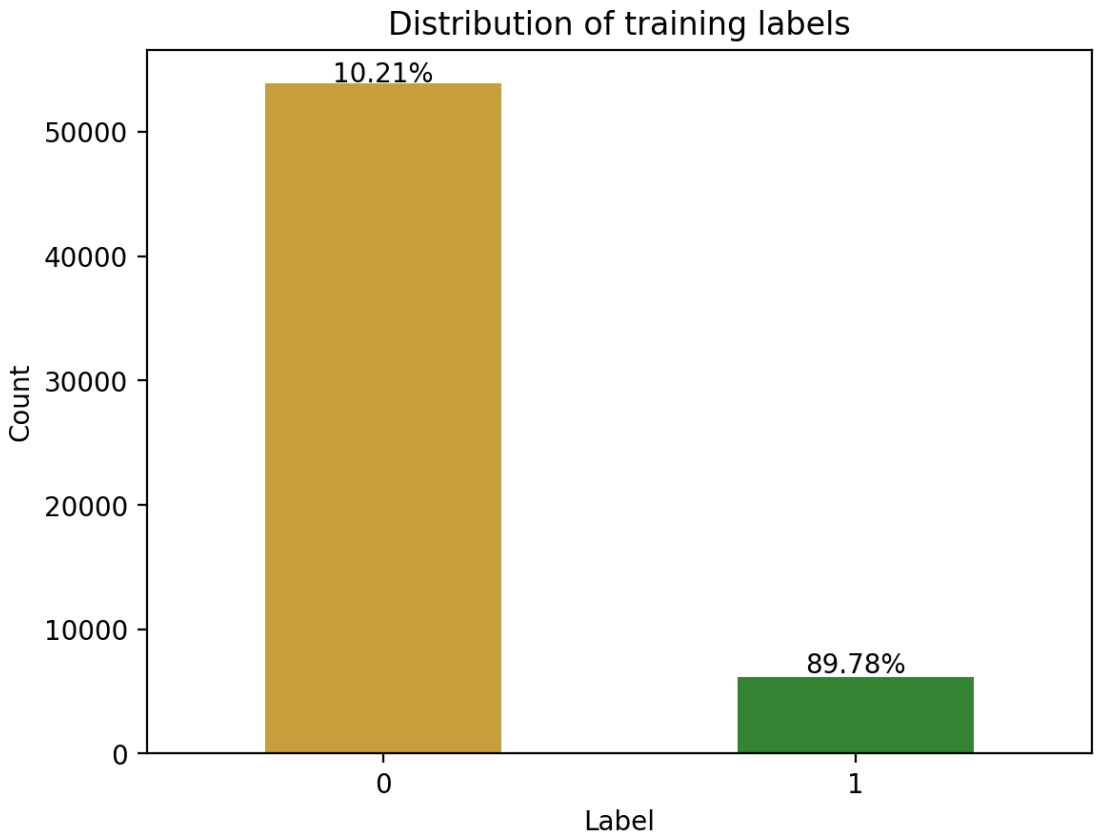
(a) Plot 10 examples of each class from the training dataset.

In [ ]:
```
# plot 10 random images of class 1 and 0
def plot_training_examples(label_class):
    '''
    Plot 10 random images of a given class
    Input: label_class: 0 or 1
    Output: None
    '''
    fig, ax = plt.subplots(2, 5)
    fig.set_size_inches(10, 4)
    fig.suptitle(f"10 random images of class {label_class}")
    fig.tight_layout()
    for i in range(2):
        np.random.seed(i)
        for j in range(5):
            X_ = X_train[y_train==label_class]
            random_index = np.random.randint(0, len(X_))
            X_train_plot = X_[random_index].reshape(28, 28)
            ax[i, j].imshow(X_train_plot, cmap='gray')
    plt.show()

plot_training_examples(1)
plot_training_examples(0)
```



(b) Plot the distribution of samples by class.

In [ ]:
```
# plot the distribution of the training labels
df_y_train = pd.DataFrame(y_train, columns=['label'])
label1 = df_y_train['label'].value_counts()[1]
label0 = df_y_train['label'].value_counts()[0]
label1_percent = label1 / (label1 + label0) * 100
label0_percent = label0 / (label1 + label0) * 100

# start plotting
df_y_train['label'].value_counts().plot(kind='bar', \
    color=['darkgoldenrod','darkgreen'], alpha = 0.8)
plt.xlabel('Label')
plt.ylabel('Count')
plt.xticks(rotation=0)
plt.text(0, 54000, f"{label1_percent:.2f}%", ha='center')
plt.text(1, 6500, f"{label0_percent:.2f}%", ha='center')
plt.title('Distribution of training labels')
plt.show()

# print out the number and percentage of class 1 and class 0
print(f"There are {label1} training examples of class 1.")
print(f"There are {label0} training examples of class 0.")
print(f"{label1_percent:.2f}% of training examples are class 1.")
print(f"{label0_percent:.2f}% of training examples are class 0.")
```




```
There are 6129 training examples of class 1.
There are 53871 training examples of class 0.
10.21% of training examples are class 1.
89.78% of training examples are class 0.
```

Among 60000 samples in the training set, 6129 samples are labeled as class 1, which is 10.21% of the total samples. On ther other hand, 53871 samples are labeled as class 0, which is 89.79% of the total samples. This is an imbalanced dataset. It may leed to problems like biased classification, as the model may be trained to primarily predict the majority class and may not perform well on the minority class.

(c) Identify the value of the regularization parameter that optimizes model performance on out-of-sample data.

- The number of model parameters that are estimated to be nonzero
- The cross entropy loss
- Area under the ROC curve (AUC)
- The F_1-score
- Which value of C seems best for this problem? (the closest power of 10)

In [ ]:
```
from sklearn.linear_model import LogisticRegression as LR
from sklearn.ensemble import RandomForestClassifier as RFC
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis as LDA
from sklearn.metrics import log_loss, f1_score, roc_auc_score, roc_curve, precision_recall_
```

In [ ]:
```
# list to store the results
l_C = np.logspace(-4, 4, 21)
l_non0_coef = []
l_loss = []
l_auc = []
l_f1 = []
```

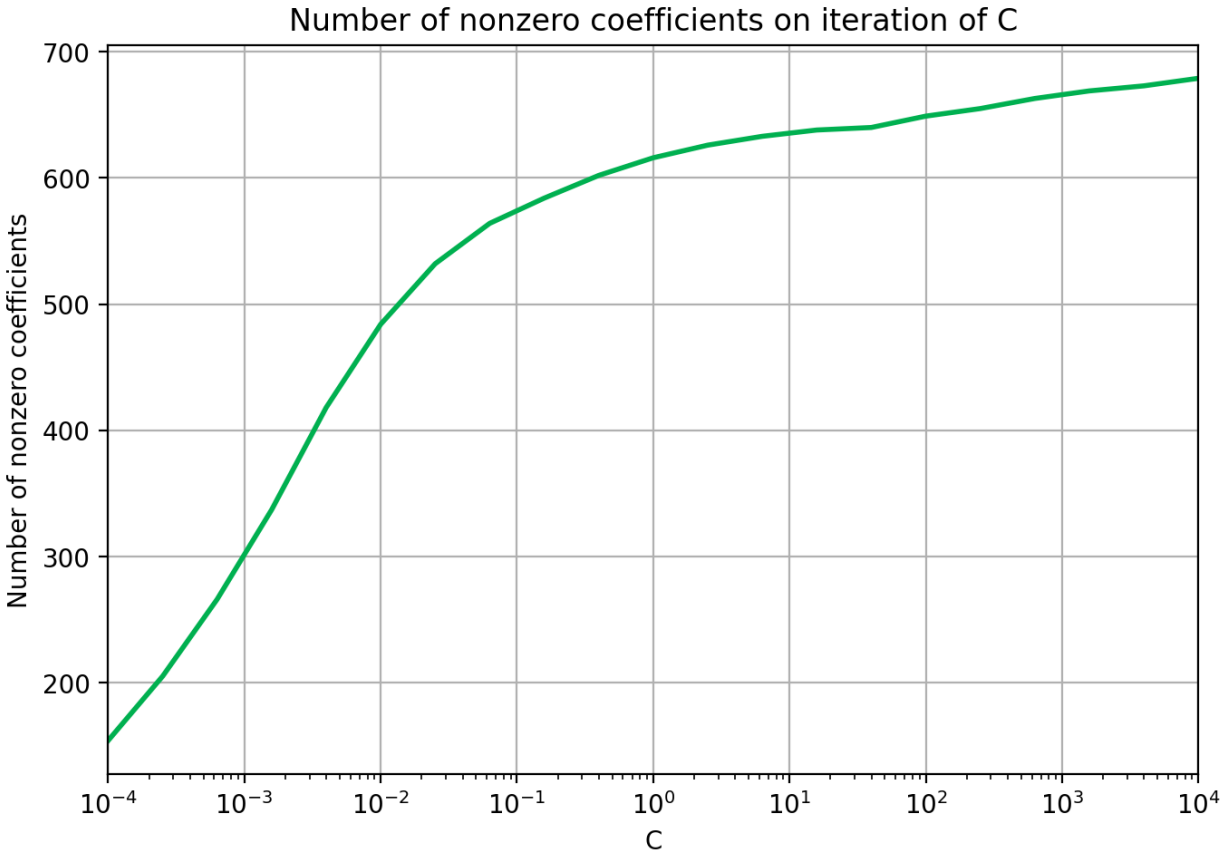
```
# start training and testing
for c in l_C:
    lr_2c = LR("l1", C=c, solver='liblinear', random_state=42).fit(X_train, y_train)
    y_pred = lr_2c.predict(X_test)
    y_pred_prob = lr_2c.predict_proba(X_test)
    # append to lists
    l_non0_coef.append((lr_2c.coef_ != 0).sum())
    l_loss.append(log_loss(y_test, y_pred))
    l_auc.append(roc_auc_score(y_test, y_pred_prob[:, 1]))
    l_f1.append(f1_score(y_test, y_pred))
```

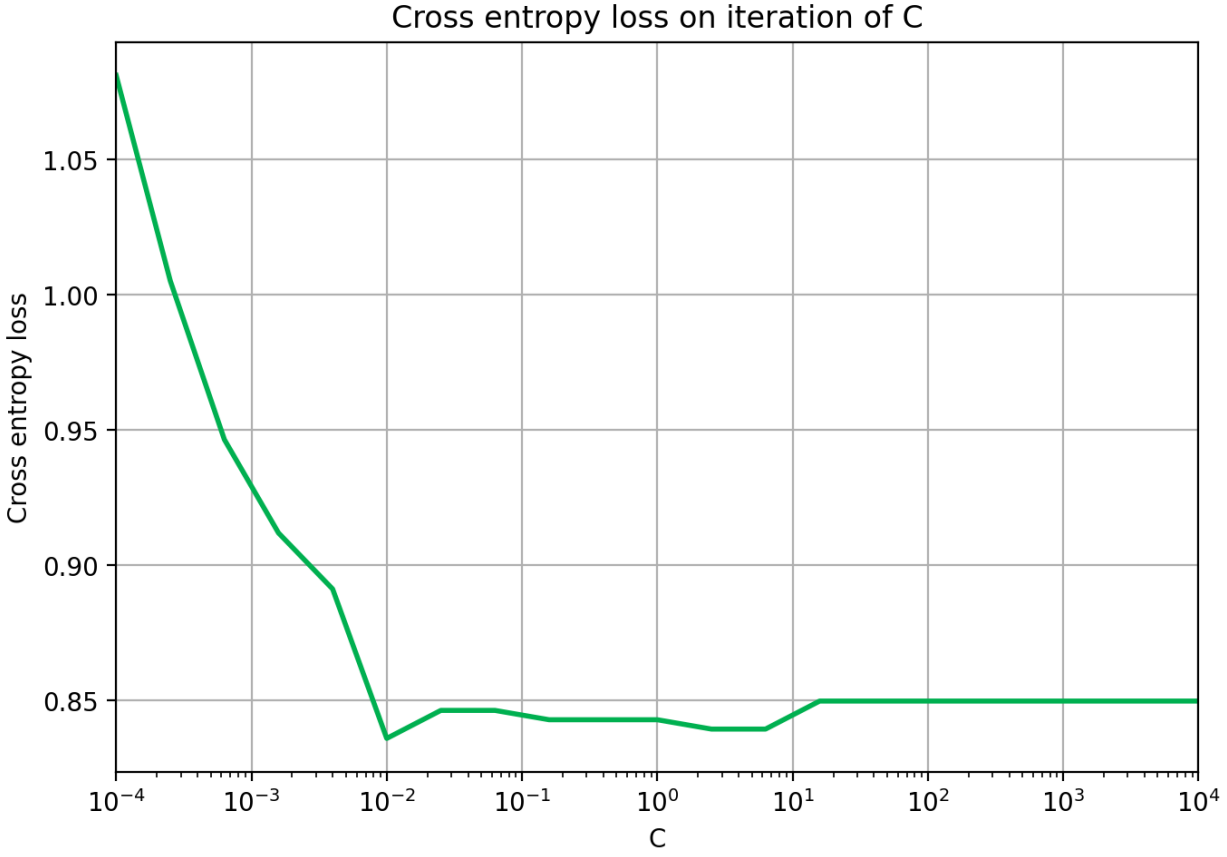
In [ ]:

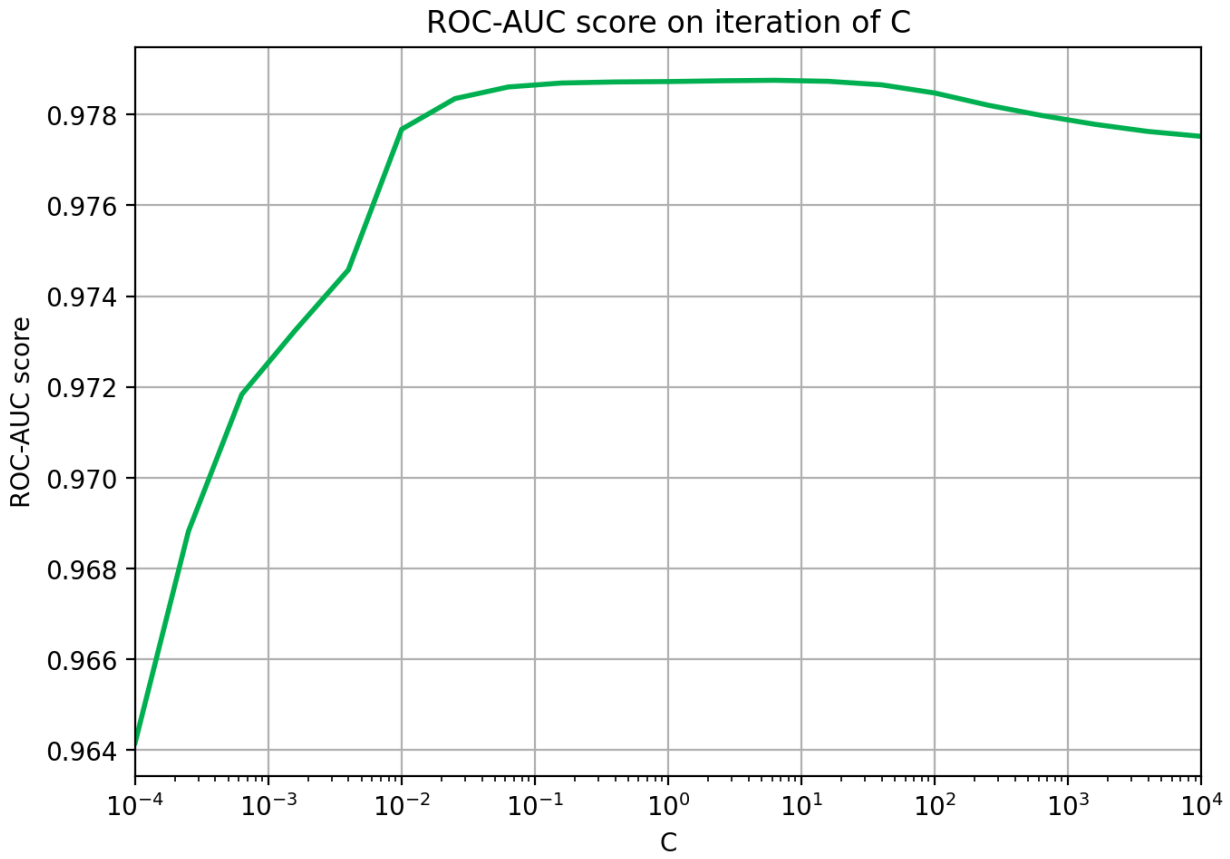
```
# define plotting function
def plot_metrics(l_metric, str_metric):
    '''
    Plot the metric on iteration of C
    Input: l_metric: list of metric values
           str_metric: string of metric name
    Output: None
    '''
    plt.figure(figsize=(7,5))
    plt.grid("on")
    plt.semilogx(l_C, l_metric, color='#00B050',linewidth=2)
    plt.xlim(l_C.min(), l_C.max())
    plt.xlabel("C")
    plt.ylabel(str_metric)
    plt.title(f"{str_metric} on iteration of C")
    plt.tight_layout()
    plt.show()
```

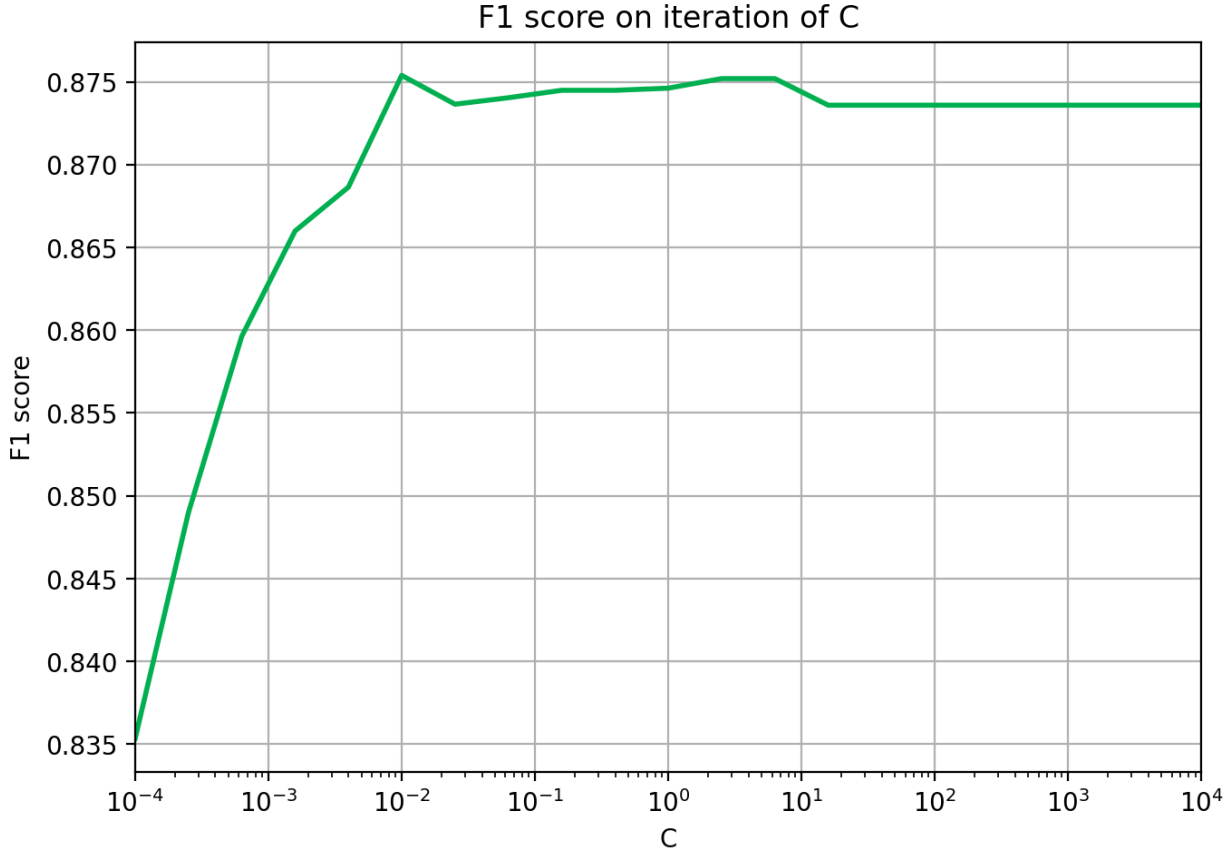
In [ ]:

```
# plot the metrics
plot_metrics(l_non0_coef, 'Number of nonzero coefficients')
plot_metrics(l_loss, 'Cross entropy loss')
plot_metrics(l_auc, 'ROC-AUC score')
plot_metrics(l_f1, 'F1 score')
```


Number of nonzero coefficients on iteration of C
Number of nonzero coefficients
700
600
500
400
300
200
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
C


Cross entropy loss on iteration of C
Cross entropy loss
1.05
1.00
0.95
0.90
0.85
10^{-4}
10^{-3}
10^{-2}
10^{-1}
10^{0}
10^{1}
10^{2}
10^{3}
10^{4}
C





Best value of C is around 0.01. I chose $C = 0.01$ because cross entropy loss reaches its minimum at this point, roc_auc and F1 score reaches their maximum at this point, and the number of nonzero parameters is around 500, which is a moderate number.

(d) Train and test

1. a logistic regression classifier with minimal regularization (C=1e100)
2. a logistic regression classifier with the best value of the regularization parameter from the last section
3. a Linear Discriminant Analysis (LDA) Classifier (using default parameters)
4. a Random Forest (RF) classifier (using default parameters)

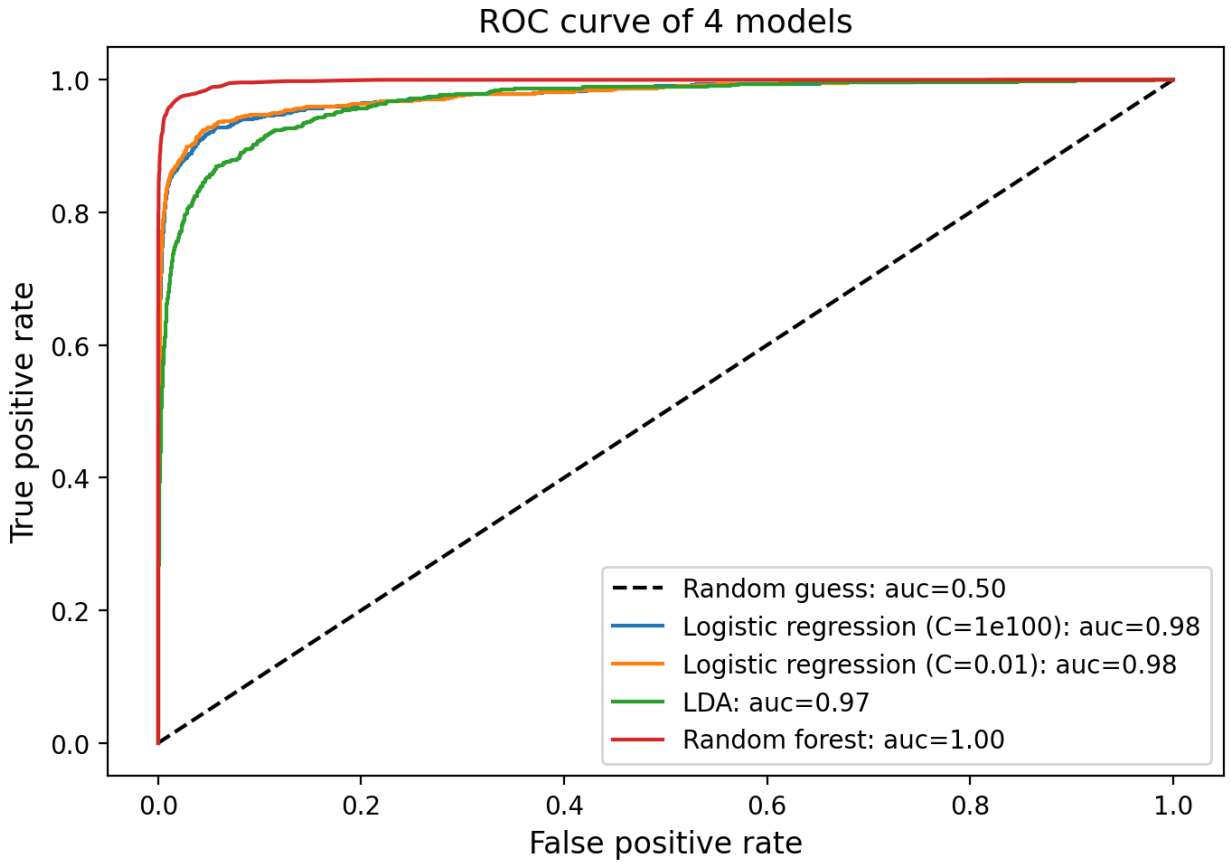
In [ ]:
```
# train 4 models
lr_minR = LR("l1", C=1e100, solver='liblinear', random_state=30).fit(X_train, y_train)
lr_bestR = LR("l1", C=0.01, solver='liblinear', random_state=42).fit(X_train, y_train)
lda = LDA().fit(X_train, y_train)
rfc = RFC(random_state=42).fit(X_train, y_train)
```

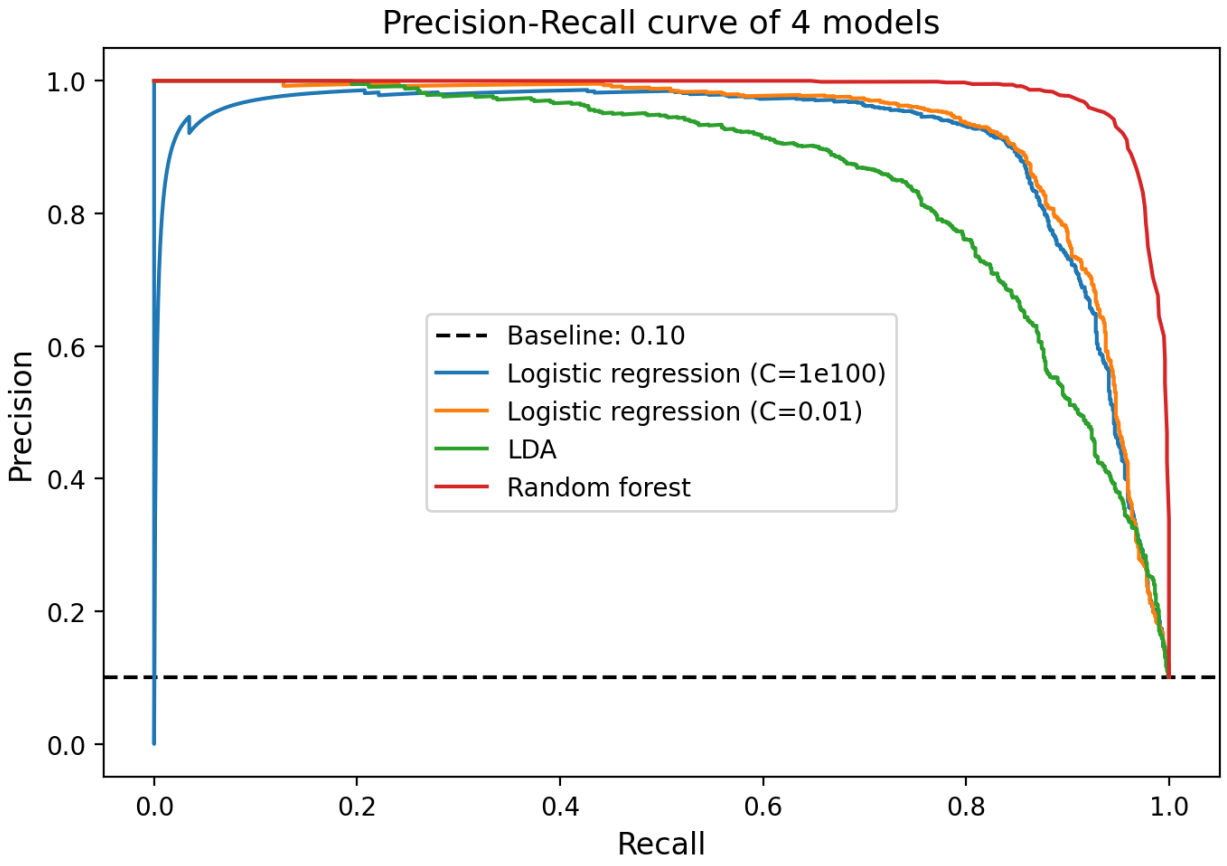
- Compare your classifiers' performance using ROC and Precision Recall (PR) curves.
- Plot the line that represents randomly guessing the class.
- Source: https://classeval.wordpress.com/introduction/introduction-to-the-precision-recall-plot/

In [ ]:
```
# iterate through the 4 models and store info for plotting
l_fpr = []
l_tpr = []
l_auc = []
l_precision = []
l_recall = []
for model in [lr_minR, lr_bestR, lda, rfc]:
    y_pred_prob = model.predict_proba(X_test)[:,1]
    # store info for plotting roc-auc curve
    fpr_, tpr_, _ = roc_curve(y_test, y_pred_prob)
    l_fpr.append(fpr_)
    l_tpr.append(tpr_)
    auc_ = roc_auc_score(y_test, y_pred_prob)
    l_auc.append(auc_)
    # store info for plotting pr curve
    precision, recall, _ = precision_recall_curve(y_test, y_pred_prob)
    l_precision.append(precision)
    l_recall.append(recall)
```

In [ ]:
```
l_model_name = ['Logistic regression (C=1e100)','Logistic regression (C=0.01)',
                'LDA', 'Random forest']
# roc-auc curve
plt.figure(figsize=(7, 5), dpi=100)
plt.plot([0, 1], [0, 1], 'k--', label='Random guess: auc=0.50')
for fpr, tpr, auc, model_name in zip(l_fpr, l_tpr, l_auc,l_model_name):
    plt.plot(fpr, tpr, label=f"{model_name}: auc={auc:.2f}")
    plt.xlabel('False positive rate', fontsize=12)
    plt.ylabel('True positive rate', fontsize=12)
    plt.title('ROC curve of 4 models', fontsize=13)
    plt.legend()
plt.tight_layout()
plt.show()

# pr curve
plt.figure(figsize=(7, 5), dpi=100)
pr_baseline = (y_test==1).sum() / len(y_test)
plt.axhline(pr_baseline, ls='--', color='k', label='Baseline: 0.10')
for precision, recall, model_name in zip(l_precision, l_recall, l_model_name):
```

```
    plt.plot(recall, precision, label=f"{model_name}")
    plt.xlabel('Recall', fontsize=12)
    plt.ylabel('Precision', fontsize=12)
    plt.title('Precision-Recall curve of 4 models', fontsize=13)
    plt.legend()
plt.tight_layout()
plt.show()
```





- 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.

According to roc-auc and precision-recall curves, random forest classifier is the best classifier for this problem. It has the highest roc-auc score and the highest precision-recall score. It is followed by logistic regression with $C = 1e100$ and $C = 0.01$. LDA classifier has the lowest model performance.

Regularization of the logistic regression doesn'e make much difference here, since both roc-auc and PR curves are very similar. We can barely tell the difference from the plots. However, when C is $1e100$, the precision and recall values fluctuate a lot in the early stage of training, causing PR curve to oscillate.

I would select random forest classifier for application to unseen data since it has the highest roc-auc score and the highest precision-recall score.