CS224 - Fall 2024 - PROGRAMMING ASSIGNMENT 2 - LINEAR AND LOGISTIC REGRESSION

Due: November 27, 2024 @ 11:59pm PDT

Maximum points: 15

Overview

In this assignment you will implement and test two supervised learning algorithms: linear regression (Question 1) and logistic regression (Question 2).

For this assignment we will use the functionality of Pandas, Matplotlib, and Numpy.

If you are asked to **implement** a particular functionality, you should **not** use an existing implementation from the libraries above (or some other library that you may find). When in doubt, please ask.

Before you start, make sure you have installed all those packages in your local Jupyter instance.

Read **all** cells carefully and answer **all** parts (both text and missing code). You will complete all the code marked `TODO`, and/or `YOUR CODE HERE`, and answer descriptive/derivation questions.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
# import random as rand
from sklearn.model_selection import train_test_split
```

Question 1: Linear Regression [10 points]

We will implement linear regression using direct solution and gradient descent.

We will first attempt to predict output using a single attribute/feature. Then we will perform linear regression using multiple attributes/features.

(a) Getting data [1 point]

In this assignment we will use the Boston housing dataset.

The Boston housing data set was collected in the 1970s to study the relationship between house price and various factors such as the house size, crime rate, socio-economic status, etc. Since the

variables are easy to understand, the data set is ideal for learning basic concepts in machine learning. The raw data and a complete description of the dataset can be found on the UCI website: https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.names, https://archive.ics.uci.edu/ml/machine-learning-databases/housing/housing.data

or

http://www.ccs.neu.edu/home/vip/teach/MLcourse/data/housing_desc.txt

We have supplied a list `names` of the column headers. When loading the data, configure `read_csv` to properly separate columns and apply the headers.

```
names =[
    'CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM',
    'AGE',  'DIS', 'RAD', 'TAX', 'PTRATIO', 'B', 'LSTAT', 'PRICE'
]

df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/housing/housing.data',
                 header=None, sep='\s+', names=names, na_values='?')
```

Create a response vector `y` with the values in the column `PRICE`. The vector `y` should be a 1D `numpy.array` structure.

```
y = df['PRICE']

print(y.shape)
print(y)
```

```
(506,)
0      24.0
1      21.6
2      34.7
3      33.4
4      36.2
       ...
501    22.4
502    20.6
503    23.9
504    22.0
505    11.9
Name: PRICE, Length: 506, dtype: float64
```

Use the response vector `y` to find the mean house price in thousands and the fraction of homes that are above $40k. (You may realize this is very cheap. Prices have gone up a lot since the 1970s!). Create print statements of the form (replace `a`'s and `b`'s with the number you get):

```
The mean house price is aa.bb thousand dollars.
Only a.b percent are above $40k.
```

```
mean_price =  y.mean()# YOUR CODE HERE
percentage_above_40k = (y.gt(40).sum() / y.shape[0]) * 100  # YOUR
CODE HERE

print(f"The mean house price is {mean_price:.2f} thousand dollars.")
print(f"Only {percentage_above_40k:.1f} percent are priced above
$40k.")
```

```
The mean house price is 22.53 thousand dollars.
Only 6.1 percent are priced above $40k.
```

(b) Visualizing the data [1 point]

Python's `matplotlib` has very good routines for plotting and visualizing data that closely follows the format of MATLAB programs. You can load the `matplotlib` package with the following commands.

```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

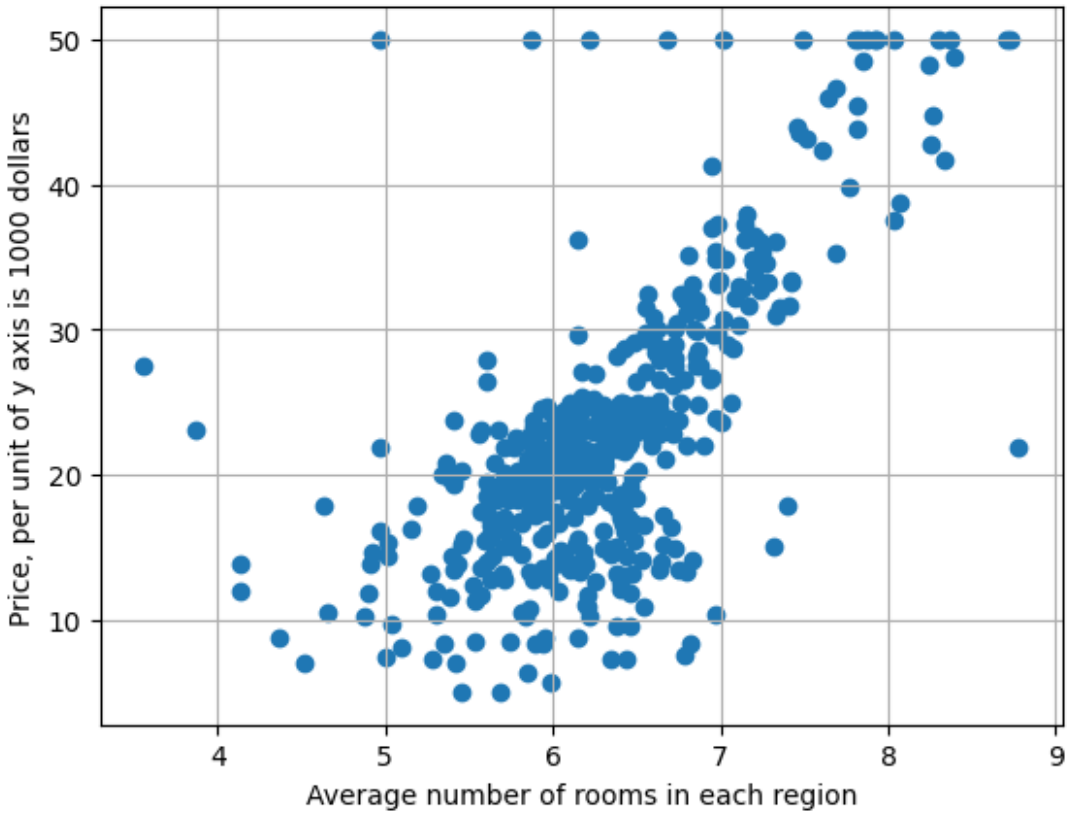
Similar to the `y` vector, create a predictor vector `x` containing the values in the `RM` column, which represents the average number of rooms in each region.

```
# TODO
x = df['RM']# YOUR CODE HERE

print(x[:3]) # print the firt 3 values of x
```

```
0    6.575
1    6.421
2    7.185
Name: RM, dtype: float64
```

Create a scatter plot of the `PRICE` vs. the `RM` attribute. Make sure your plot has **grid lines** and label the axes with reasonable **labels** so that someone else can understand the plot.

```
# YOUR CODE HERE
fig = plt.figure()
plt.scatter(x, y)
plt.grid()
plt.xlabel('Average number of rooms in each region')
plt.ylabel('Price, per unit of y axis is 1000 dollars')
plt.show()
```



The number of rooms and price seem to have a linear trend, so let us try to predict price using number of rooms first.

Derivation of a simple linear model for a single feature

Suppose we have N pairs of training samples $(x_1, y_1), \ldots, (x_N, y_N)$, where $x_i \in R$ and $y_i \in R$.

We want to perform a linear fit for this 1D data as

$$y = w\,x + b,$$

where $w \in R$ and $b \in R$.

The optimal values of $w^{¿}, b^{¿}$ that minimize the loss function

$$L(w, b) = \sum_{i=1}^{N} (w\,x_i + b - y_i)^2$$

can be written as

$$w^{¿} = \frac{\sum_i (x_i - \acute{x})(y_i - \acute{y})}{\sum_i (x_i - \acute{x})^2}$$

and

$$b^{\hat{}} = \acute{y} - w^{\hat{}}\acute{x},$$

where $\acute{x} = \frac{1}{N}\sum_i x_i, \acute{y} = \frac{1}{N}\sum_i y_i$ are mean values of x_i, y_i, respectively.

(c) Fitting a linear model using a single feature [1 point]

Use the formulae above to compute the parameters w, b in the linear model $y = w\,x + b$.

```
def fit_linear(x,y):
    """
    Given vectors of data points (x,y), performs a fit for the linear
model:
       yhat = w*x + b,
    The function returns w and b
    """
    # TODO: Calculate means of x and y
    x_mean = x.mean() # YOUR CODE HERE
    y_mean = y.mean() # YOUR CODE HERE

    # TODO: Compute w* and b* using the formulas
    # YOUR CODE HERE
    numerator = ((x - x_mean) * (y - y_mean)).sum()
    denominator = ((x - x_mean) ** 2).sum()
    w =  numerator / denominator # YOUR CODE HERE
    b = y_mean - w * x_mean# YOUR CODE HERE

    return w, b

# Using the function `fit_linear` above, print the values `w`, `b` for
the linear model of price vs. number of rooms.
w, b = fit_linear(x,y)
print(f'w = {w:.2f}, b = {b:.2f}')
```

```
w = 9.10, b = -34.67
```

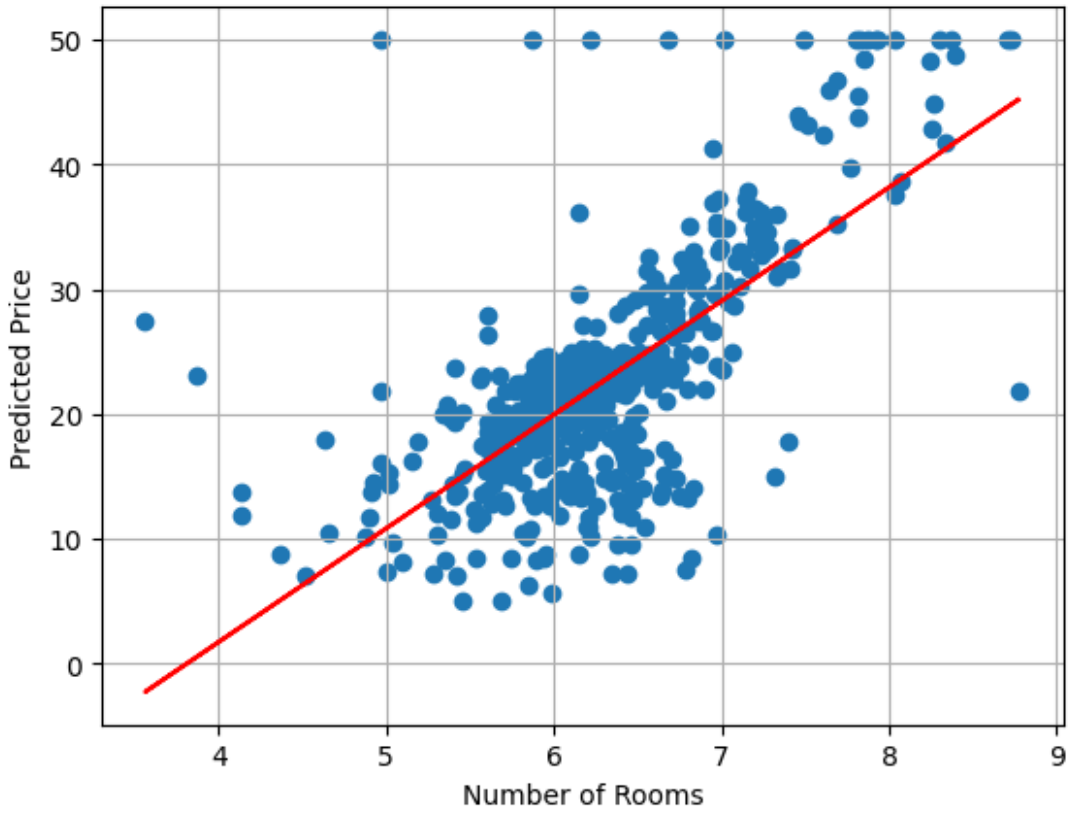
Does the price increase or decrease with the number of rooms?

- Except some outliers, the price on an average increases if the number of rooms increase

Replot the scatter plot above, but now with the regression line.

```
# Prediction function using the optimal parameters
def predict(x, w, b):
    return w * x + b # Prediction function using the optimal
parameters

# Points on the regression line
y_pred = predict(x, w, b)
```

```
plt.xlabel('Number of Rooms')
plt.ylabel('Predicted Price')
plt.scatter(x, y)
plt.plot(x, y_pred, c='r')
plt.grid()
```



(d) Linear regression with multiple features/attributes [2 points]

One possible way to try to improve the fit is to use multiple variables at the same time.

In this problem, the target variable will still be the `PRICE`. We will use multiple attributes of the house to predict the price.

The names of all the data attributes are given in variable `names`.

- We can get the list of names of the columns from `df.columns.tolist()`.
- Remove the last items from the list using indexing.

```
xnames = names[:-1]
print(names[:-1])
```

```
['id', 'thick', 'size_unif', 'shape_unif', 'marg', 'cell_size',
'bare', 'chrom', 'normal', 'mit']
```

Let us use `CRIM`, `RM`, and `LSTAT` to predict `PRICE`.

Get the data matrix `X` with three features (`CRIM`, `RM`, `LSTAT`) and target vector `y` from the dataframe `df`.

Recall that to get the items from a dataframe, you can use syntax such as

```
s = np.array(df['RM'])
```

which gets the data in the column `RM` and puts it into an array `s`. You can also get multiple columns with syntax like

```
X12 = np.array(df[['CRIM', 'ZN']])

# TODO
X = np.array(df[['CRIM', 'RM', 'LSTAT']]) # YOUR CODE HERE

print(X.shape)
```

```
(506, 3)
```

Linear regression in scikit-learn

To fit the linear model, we could create a regression object and then fit the training data with regression object.

```
from sklearn import linear_model
regr = linear_model.LinearRegression()
regr.fit(X_train,y_train)
```

You can see the coefficients as

```
regr.intercept_
regr.coef_
```

We can predict output for any data as

```
y_pred = regr.predict(X)
```

Instead of taking this approach, we will implement the regression function on our own.

Split the Data into Training and Test

Split the data into training, validation, and testing set. Use 70% for training and 30% for testing. You can do the splitting manually or use the `sklearn` package `train_test_split`. Store the training data in `X_train, y_train` and test data in `X_test, y_test`.

```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=20)# YOUR CODE HERE
```

```
print('Shape of X_train: ', X_train.shape)
print('Shape of X_test: ', X_test.shape)
print('Shape of y_train: ', y_train.shape)
print('Shape of y_test: ', y_test.shape)
```

```
Shape of X_train:  (354, 3)
Shape of X_test:  (152, 3)
Shape of y_train:  (354,)
Shape of y_test:  (152,)
```

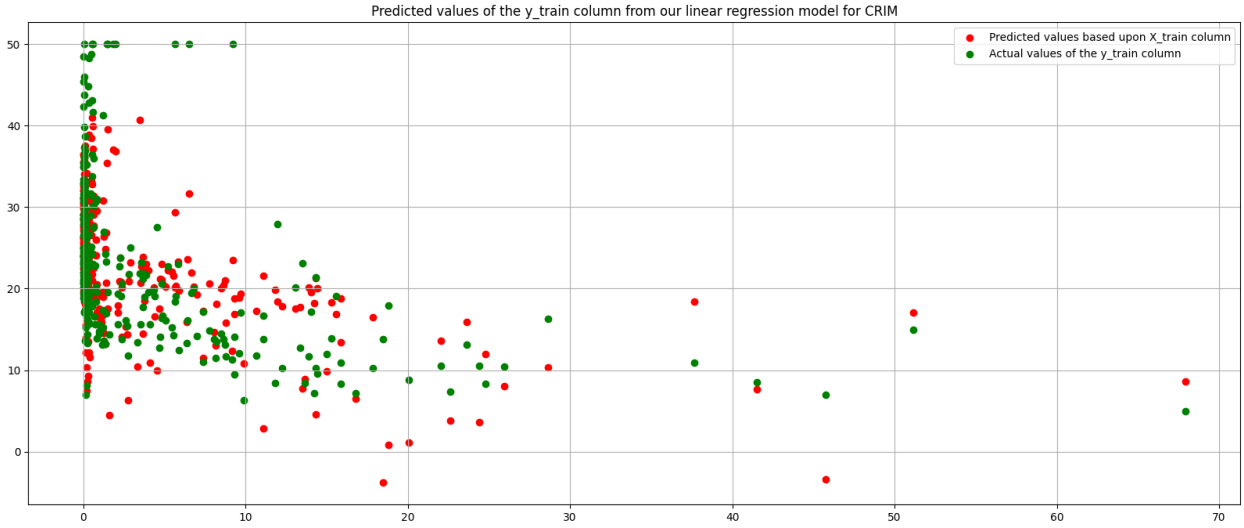
Compute the predicted values `yhat_tr` on the training data and print the average square loss value on the **training** data.

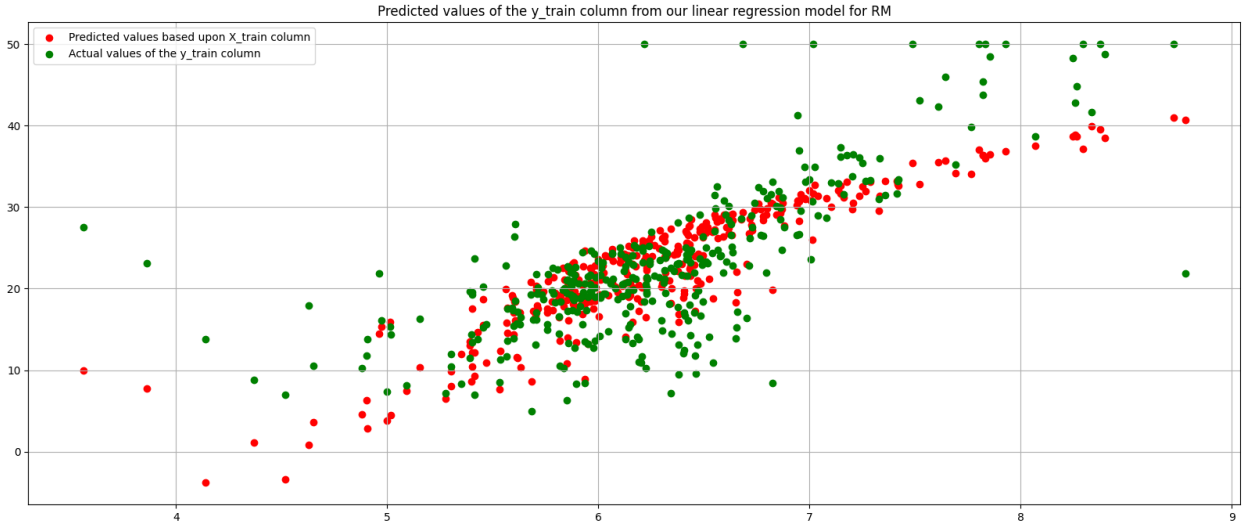
```
# YOUR CODE HERE
from sklearn import linear_model
regr = linear_model.LinearRegression()
regr.fit(X_train, y_train)
y_pred = regr.predict(X_train)
print(((y_train - y_pred)**2).sum())
```

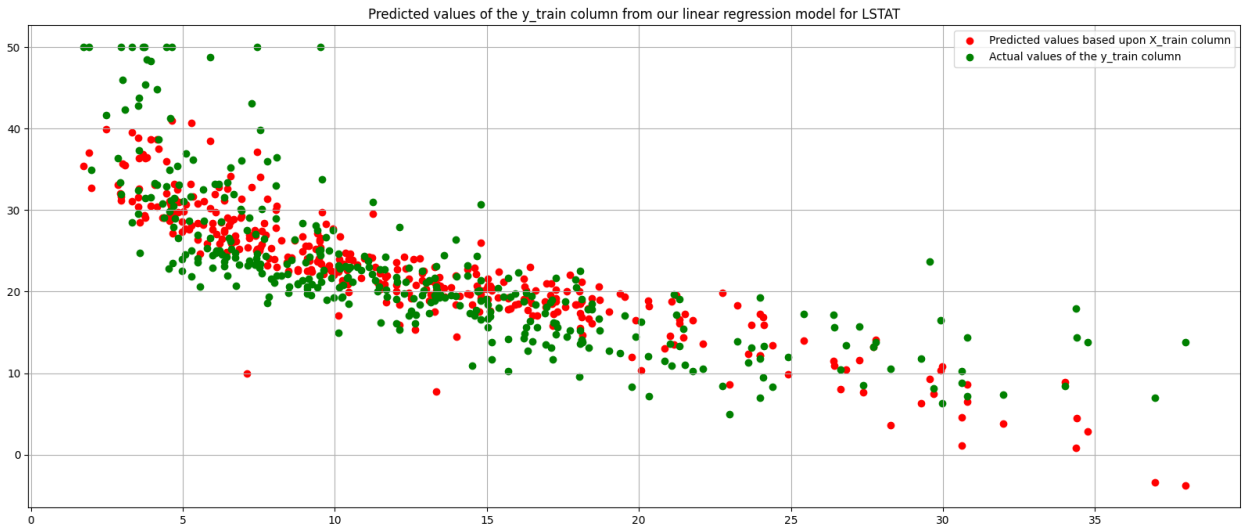
```
9731.795002349414
```

Create a scatter plot of the actual vs. predicted values of `y` on the **training** data.

```
# YOUR CODE HERE
# Different 2d plots based upon different features
for i, f in enumerate(['CRIM', 'RM', 'LSTAT']):
    fig = plt.figure(figsize=(20, 8))
    plt.scatter(X_train[:,i], y_pred, color='r', label='Predicted
values based upon X_train column')
    plt.scatter(X_train[:,i], y_train, color='g', label='Actual values
of the y_train column')
    # plt.xlabel('Actual values of the y_train column')
    # # plt.ylabel('Predicted values of the y_train column from our
linear regression model')
    plt.title(f'Predicted values of the y_train column from our linear
regression model for {f}')
    plt.legend()
    plt.grid()
    plt.show()
```


Predicted values of the y_train column from our linear regression model for CRIM
Predicted values based upon X_train column
Actual values of the y_train column


Predicted values of the y_train column from our linear regression model for RM
Predicted values based upon X_train column
Actual values of the y_train column


Predicted values of the y_train column from our linear regression model for LSTAT
Predicted values based upon X_train column
Actual values of the y_train column

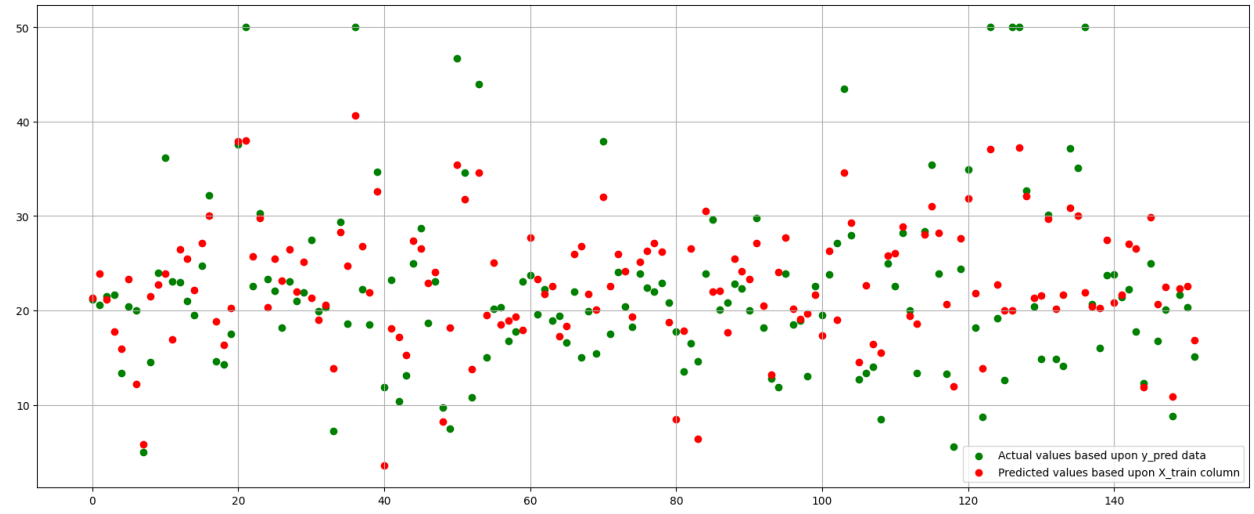
Compute the predicted values `yhat_ts` on the test data and print the average square loss value on the **testing** data.

```
# YOUR CODE HERE
y_pred_test = regr.predict(X_test)
print(((y_pred_test - y_test)**2).sum())
# fig = plt.figure(figsize=(20, 8))
# plt.scatter(y_test, y_pred_test)
# plt.xlabel('Actual values of the y_test column')
# plt.ylabel('Predicted values of the y_test column from our linear
regression model')
# plt.grid()
# plt.show()
```

```
5512.686914934722
```

Create a scatter plot of the actual vs. predicted values of `y` on the **testing** data.

```
# YOUR CODE HERE

fig = plt.figure(figsize=(20, 8))
xs = [x for x in range(len(y_test))]
plt.scatter(xs, y_test, color='g', label='Actual values based upon
y_pred data')
plt.scatter(xs, y_pred_test, color='r', label='Predicted values based
upon X_train column')
# plt.xlabel('Actual values of the y_test column')
# plt.ylabel('Predicted values of the y_test column from our linear
regression model')
plt.legend()
plt.grid()
plt.show()
```



Gradient descent for linear regression [2 points]

Finally, we will implement the gradient descent version of linear regression.

In particular, the function implemented should follow the following format:

```
def linear_regression_gd(X,y,learning_rate =
0.00001,max_iter=10000,tol=pow(10,-5), regularization_param=0.0):
```

where `X` is the same data matrix used above (with ones column appended), `y` is the variable to be predicted, `learning_rate` is the learning rate used (α or ρ_t in the slides), `max_iter` defines the maximum number of iterations that gradient descent is allowed to run, `tol` is defining the tolerance for convergence (which we'll discuss next), and `regularization_param` is the hyperparameter λ for L2 regularization, which helps to prevent overfitting.

The return values for the above function should be (at the least) 1) `w` which are the regression parameters, 2) `all_cost` which is an array where each position contains the value of the objective function $L(w)$ for a given iteration, 3) `iters` which counts how many iterations did the algorithm need in order to converge to a solution.

Gradient descent is an iterative algorithm; it keeps updating the variables until a convergence criterion is met. In our case, our convergence criterion is whichever of the following two criteria happens first:

- The maximum number of iterations is met
- The relative improvement in the cost is not greater than the tolerance we have specified. For this criterion, you may use the following snippet into your code:

```
np.absolute(all_cost[it] - all_cost[it-1])/all_cost[it-1] <= tol

# TODO
# Implement gradient descent for linear regression

def compute_cost(X,w,y, reg):
    # YOUR CODE HERE
    m = len(y)
    error = X.dot(w) - y
    L = (1/(2*m)) * np.sum(error ** 2)
    if reg > 0.0:
        reg = (reg / 2 * m) * np.sum(w[1:] ** 2)
        L = L + reg
    return L

def linear_regression_gd(X,y,learning_rate =
0.00001,max_iter=10000,tol=pow(10,-5), regularization_param=0.0):
    # YOUR CODE HERE
    iters = 0
    m, n = X.shape
    w = np.zeros(n)
    all_cost = []
```

```
    for it in range(max_iter):
        iters = iters + 1
        error = X.dot(w) - y
        gradient = (1/m) * X.T.dot(error)
        if regularization_param > 0.0:
            gradient[1:] += (regularization_param / m) * w[1:]
        w = w - learning_rate * gradient
        loss = compute_cost(X, w, y, reg=regularization_param)
        all_cost.append(loss)
        if (it > 1) and (np.absolute(all_cost[it] - all_cost[it-
1])/all_cost[it-1] <= tol):
            break
    return w,all_cost,iters
```

K-Fold Cross-Validation [2 ponts]

Now, let's implement k-fold cross-validation (k=5) to evaluate the performance of your gradient descent function with the regularization hyperparameter.

1. Split the dataset into k equal parts (folds).
2. For each fold, use the current fold as the test set and the remaining k-1 folds as the training set.
3. Record the final cost from the training process.
4. After training on all folds, calculate the average cost across all k folds.
5. Assess the models' performance on testing set.

```
k = 5
fold_size = len(X_train) // k
indices = np.random.permutation(len(X))
fold_costs = []
for fold in range(k):
    test_indices = indices[fold * fold_size: (fold + 1) * fold_size]
# Indices for the test set
    train_indices = np.setdiff1d(indices, test_indices)  # Indices for
the training set

    X_train_fold = X[train_indices]
    y_train_fold = y[train_indices]
    X_test_fold = X[test_indices]
    y_test_fold = y[test_indices]
    w, cost, iters = linear_regression_gd(X_train_fold, y_train_fold)
# rest parameters are default
    test_cost = compute_cost(X_test_fold, w, y_test_fold, 0.0)
    fold_costs.append(test_cost)
    print("Done for current fold, cost is ", test_cost, " and iters is
", iters)

avg_cost = np.mean(fold_costs)
```

```
print(f"Average cost over {k} folds: {avg_cost}")
# YOUR CODE HERE
```

```
Done for current fold, cost is  49.04773842670627  and iters is  10000
Done for current fold, cost is  23.621529266888913  and iters is
10000
Done for current fold, cost is  37.93761592694733  and iters is  10000
Done for current fold, cost is  11.180441880542604  and iters is
10000
Done for current fold, cost is  16.569489483224935  and iters is
10000
Average cost over 5 folds: 27.67136299686201
```

Convergence plots [1 points]

After implementing gradient descent for linear regression, we would like to test that indeed our algorithm converges to a solution. In order see this, we are going to look at the value of the objective/loss function $L(w)$ as a function of the number of iterations, and ideally, what we would like to see is $L(w)$ drops as we run more iterations, and eventually it stabilizes.

The learning rate plays a big role in how fast our algorithm converges: a larger learning rate means that the algorithm is making faster strides to the solution, whereas a smaller learning rate implies slower steps. In this question we are going to test two different values for the learning rate:

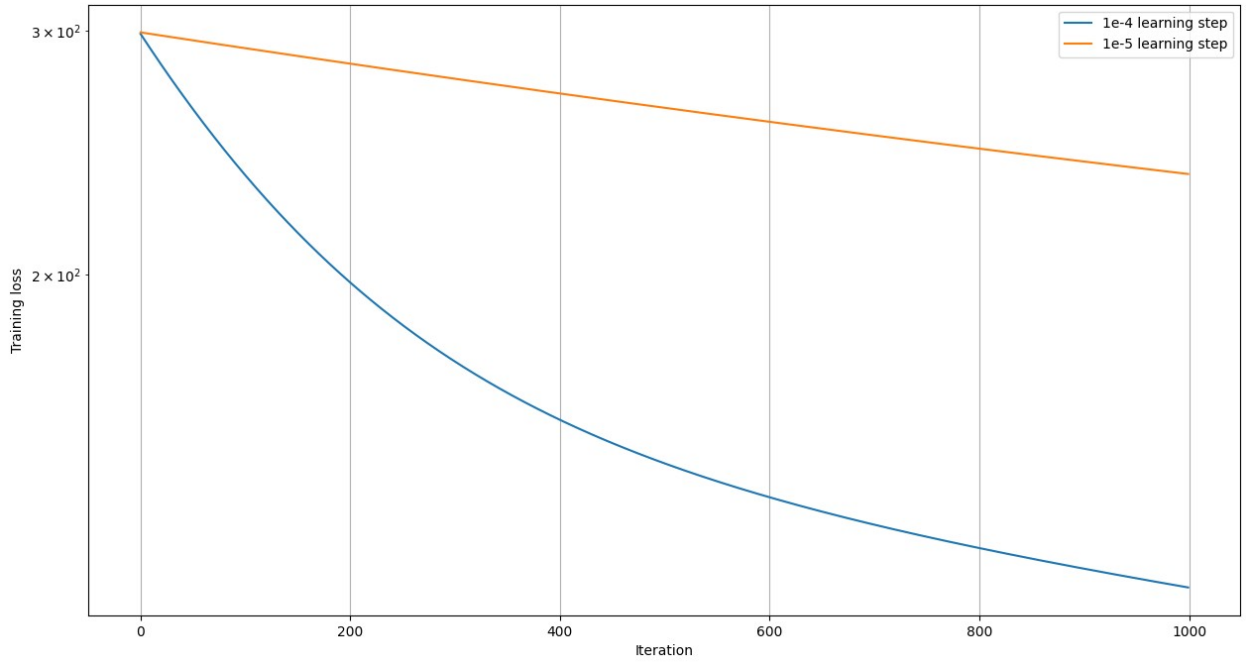
- 0.00001
- 0.000001

while keeping the default values for the max number of iterations and the tolerance.

- Plot the two convergence plots (cost(loss) vs. iterations)
- What do you observe?

```
# TODO
# test gradient descent with step size 0.00001
(w, all_cost,iters) =
linear_regression_gd(X_train,y_train,learning_rate = 0.00001,max_iter
= 1000, tol=pow(10,-6), regularization_param=0.0)#TODO|
plt.figure(figsize=(15, 8))
plt.semilogy(all_cost[0:iters], label='1e-4 learning step')


# test gradient descent with step size 0.000001
# complete the rest
# YOUR CODE HERE
(w, all_cost,iters) =
linear_regression_gd(X_train,y_train,learning_rate = 0.000001,max_iter
= 1000, tol=pow(10,-6), regularization_param=0.0)#TODO|
plt.semilogy(all_cost[0:iters], label='1e-5 learning step')
```

```
plt.grid()
plt.xlabel('Iteration')
plt.ylabel('Training loss')
plt.legend()
plt.show()
```



Observations:

- Small learning step is even after 1000 iterations is still at a very high loss.
- The tolerance value is too small so all iterations are being used up, however we do see an exponential loss reduction.
- When the steps to 10000, it shows some stabilization of the loss function.

Question 2. Logistic Regression [5 points]

In this question, we will plot the logistic function and perform logistic regression. We will use the breast cancer data set. This data set is described here: https://archive.ics.uci.edu/dataset/15/breast+cancer+wisconsin+original.

Each sample is a collection of features that were manually recorded by a physician upon inspecting a sample of cells from fine needle aspiration. The goal is to detect if the cells are benign or malignant.

We could use the `sklearn` built-in `LogisticRegression` class to find the weights for the logistic regression problem. The `fit` routine in that class has an *optimizer* to select the weights to best match the data. To understand how that optimizer works, in this problem, we will build a very simple gradient descent optimizer from scratch.

Loading and visualizing the Breast Cancer Data

We load the data from the UCI site and remove the missing values.

```
names =
['id','thick','size_unif','shape_unif','marg','cell_size','bare',
         'chrom','normal','mit','class']
df = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-
databases/' +
                 'breast-cancer-wisconsin/breast-cancer-
wisconsin.data',
                 names=names,na_values='?',header=None)
df = df.dropna()
df.head(6)
```

	id	thick	size_unif	shape_unif	marg	cell_size	bare	chrom	normal	mit	class
0	1000025	5	1	1	1	2	1.0	3	1	1	2
1	1002945	5	4	4	5	7	10.0	3	2	1	2
2	1015425	3	1	1	1	2	2.0	3	1	1	2
3	1016277	6	8	8	1	3	4.0	3	7	1	2
4	1017023	4	1	1	3	2	1.0	3	1	1	2
5	1017122	8	10	10	8	7	10.0	9	7	1	4

```
len(df)
```

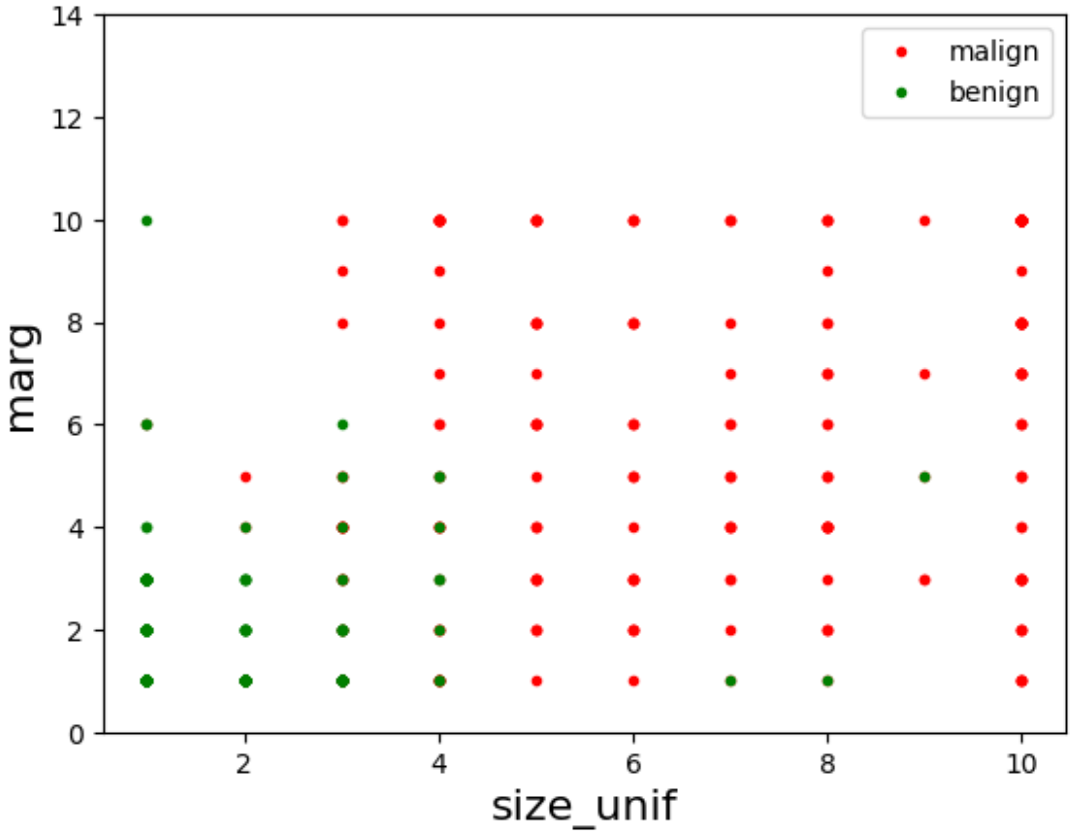
```
683
```

After loading the data, we can create a scatter plot of the data labeling the class values with different colors. We will pick two of the features.

```
# Get the response.  Convert to a zero-one indicator
yraw = np.array(df['class'])
BEN_VAL = 2   # value in the 'class' label for benign samples
MAL_VAL = 4   # value in the 'class' label for malignant samples
y = (yraw == MAL_VAL).astype(int)
```

```
Iben = (y==0)
Imal = (y==1)

# Get two predictors
xnames =['size_unif','marg']
X = np.array(df[xnames])

# Create the scatter plot
plt.plot(X[Imal,0],X[Imal,1],'r.')
plt.plot(X[Iben,0],X[Iben,1],'g.')
plt.xlabel(xnames[0], fontsize=16)
plt.ylabel(xnames[1], fontsize=16)
plt.ylim(0,14)
plt.legend(['malign','benign'],loc='upper right')
```

```
<matplotlib.legend.Legend at 0x134a653d0>
```



The above plot is not informative, since many of the points are on top of one another. Thus, we cannot see the relative frequency of points.

We see that $\sigma(w\,x+b)$ represents the probability that $y=1$. The function $\sigma(w\,x)>0.5$ for $x>0$ meaning the samples are more likely to be $y=1$. Similarly, for $x<0$, the samples are more likely to be $y=0$. The scaling w determines how fast that transition is and b influences the transition point.

Fitting the Logistic Model on Two Variables

We will fit the logistic model on the two variables `size_unif` and `marg`.

```
# load data
xnames =['size_unif','marg']
X = np.array(df[xnames])
print(X.shape)
```

```
(683, 2)
```

Next we split the data into training and test. Use 70% for training and 30% for testing. You can do the splitting manually or use the `sklearn` package `train_test_split`. Store the training data in `Xtr,ytr` and test data in `Xts,yts`.

```
X_train, X_test, y_train, y_test =  train_test_split(X, y,
test_size=0.3, random_state=20)# YOUR CODE HERE

print('Shape of X_train: ', X_train.shape)
print('Shape of X_test: ', X_test.shape)
print('Shape of y_train: ', y_train.shape)
print('Shape of y_test: ', y_test.shape)
```

```
Shape of X_train:  (478, 2)
Shape of X_test:  (205, 2)
Shape of y_train:  (478,)
Shape of y_test:  (205,)
```

Logistic regression in scikit-learn

The actual fitting is easy with the `sklearn` package. The parameter `C` states the level of inverse regularization strength with higher values meaning less regularization. Right now, we will select a high value to minimally regularize the estimate.

We can also measure the accuracy on the test data. You should get an accuracy around 90%.

```
from sklearn import datasets, linear_model, preprocessing
reg = linear_model.LogisticRegression(C=1e5)
reg.fit(X_train, y_train)

print(reg.coef_)
print(reg.intercept_)

yhat = reg.predict(X_test)
acc = np.mean(yhat == y_test)
print("Accuracy on test data = %f" % acc)
```

```
[[1.83622746 0.4450255 ]]
[-6.78742107]
Accuracy on test data = 0.926829
```

Instead of taking this approach, we will implement the regression function using gradient descent.

(a) Gradient descent for logistic regression [2 points]

The weight vector can be found by minimizing the negative log likelihood over N training samples. The negative log likelihood is called the *loss* function. For the logistic regression problem, the loss function simplifies to

$$L(w) = -\sum_{i=1}^{N} y_i \log \sigma\left(w^T x_i + b\right) + \left(1 - y_i\right) \log\left[1 - \sigma\left(w^T x_i + b\right)\right).$$

Gradient can be computed as

$$\nabla_w L = \sum_{i=1}^{N} \left(\sigma\left(w^T x_i\right) - y_i\right) x_i, \nabla_b L = \sum_{i=1}^{N} \left(\sigma\left(w^T x_i\right) - y_i\right).$$

We can update w, b at every iteration as

$$ \mathbf{w} \gets \mathbf{w} - \alpha \nabla_\mathbf{w}L, \\ b \gets b - \alpha \nabla_b L.$$

Note that we could also append the constant term in w and append 1 to every x_i accordingly, but we kept them separate in the expressions above.

Gradient descent function implementation

We will use this loss function and gradient to implement a gradient descent-based method for logistic regression.

Recall that training a logistic function means finding a weight vector `w` for the classification rule:

$$P(y = 1 \vee x, w) = \frac{1}{1 + e^{-z}}, z = w[0] + w[1] \cdot x[1] + \cdots + w[d] \cdot x[d]$$

The function implemented should follow the following format:

```
def logistic_regression_gd(X,y,learning_rate =
0.001,max_iter=1000,tol=pow(10,-5)):
```

Where `X` is the training data feature(s), `y` is the variable to be predicted, `learning_rate` is the learning rate used (α in the slides), `max_iter` defines the maximum number of iterations that gradient descent is allowed to run, and `tol` is defining the tolerance for convergence (which we'll discuss next).

The return values for the above function should be (at the least):

1. `w` which are the regression parameters,
2. `all_cost` which is an array where each position contains the value of the objective function $L(w)$ for a given iteration,

3. `iters` which counts how many iterations did the algorithm need in order to converge to a solution.

Gradient descent is an iterative algorithm; it keeps updating the variables until a convergence criterion is met. In our case, our convergence criterion is whichever of the following two criteria happens first:

- The maximum number of iterations is met
- The relative improvement in the cost is not greater than the tolerance we have specified. For this criterion, you may use the following snippet into your code:

```
np.absolute(all_cost[it] - all_cost[it-1])/all_cost[it-1] <= tol
```

```
# TODO
# Your code for logistic regression via gradient descent goes here

def compute_cost(X,w,y):
    # your code for the loss function goes here
    m = len(y)
    pred = X.dot(w)
    sigmoid = 1/(1+np.exp(-pred))
    L = (-1/m) * np.sum(y*np.log(sigmoid) + (1-y)*np.log(1-sigmoid))
    return L

def logistic_regression_gd(X,y,learning_rate = 1e-
5,max_iter=1000,tol=pow(10,-5)):
    # your code goes here
    m, n = X.shape
    w = np.zeros(n)
    all_cost = []
    iters = 0

    for it in range(max_iter):
        iters = iters + 1
        pred = X.dot(w)
        sigmoid = 1/(1+np.exp(-pred))
        gradient = (1/m) * X.T.dot(pred - y)
        w = w - learning_rate * gradient
        loss = compute_cost(X, w, y)
        all_cost.append(loss)
        if (it > 1) and (np.absolute(all_cost[it] - all_cost[it-
1])/all_cost[it-1] <= tol):
            break

    return w, all_cost,iters
```

(b) Convergence plots and test accuracy [1 point]

After implementing gradient descent for logistic regression, we would like to test that indeed our algorithm converges to a solution. In order see this, we are going to look at the value of the

objective/loss function $L(w)$ as a function of the number of iterations, and ideally, what we would like to see is $L(w)$ drops as we run more iterations, and eventually it stabilizes.

The learning rate plays a big role in how fast our algorithm converges: a larger learning rate means that the algorithm is making faster strides to the solution, whereas a smaller learning rate implies slower steps. In this question we are going to test two different values for the learning rate:

- 0.001
- 0.00001

while keeping the default values for the max number of iterations and the tolerance.

- Plot the two convergence plots (cost vs. iterations)
- Calculate the accuracy of classifier on the test data `Xts`
- What do you observe?

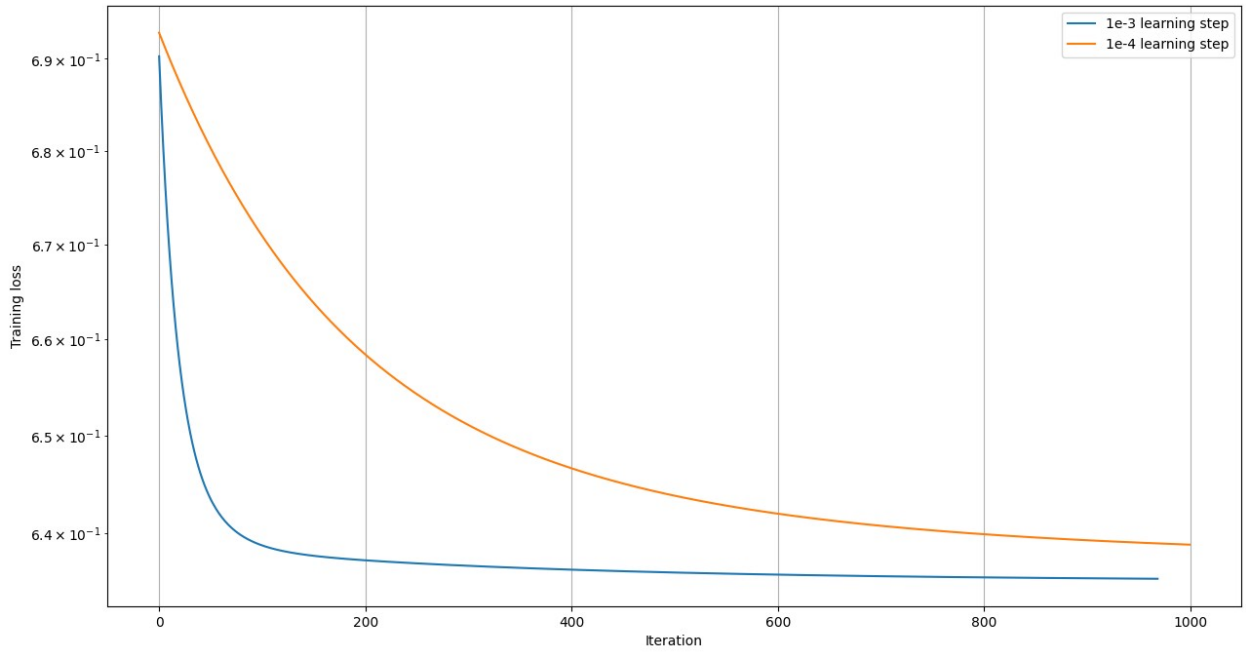
Calculate accuracy of your classifier on test data

To calculate the accuracy of our classifier on the test data, we can create a predict method.

Implement a function `predict(X,w)` that provides you label 1 if $w^T x+b>0$ and 0 otherwise.

```
# TODO: Predict on test samples and measure accuracy
def predict(X,w):
  # your code goes here
  pred = X.dot(w)
  return yhat

# TODO
# test gradient descent with step size 0.001
# test gradient descent with step size 0.00001

(w, all_cost,iters) =
logistic_regression_gd(X_train,y_train,learning_rate = 0.001,max_iter
= 1000, tol=pow(10,-6))
plt.figure(figsize=(15, 8))
plt.semilogy(all_cost[0:iters], label='1e-3 learning step')

# print(iters)
yhat = predict(X_test,w)
acc = np.mean(yhat == y_test)
print("Test accuracy for 1e-3 = %f" % acc)

# complete the rest
(w, all_cost,iters) =
logistic_regression_gd(X_train,y_train,learning_rate = 0.0001,max_iter
= 1000, tol=pow(10,-6))
plt.semilogy(all_cost[0:iters], label='1e-4 learning step')
plt.grid()
plt.xlabel('Iteration')
```

```
plt.ylabel('Training loss')
plt.legend()
plt.show()

# print(iters)
yhat = predict(X_test,w)
acc = np.mean(yhat == y_test)
print("Test accuracy for 1e-4 = %f" % acc)


Test accuracy for 1e-3 = 0.926829
```




```
Test accuracy for 1e-4 = 0.926829
```

Observations:

- 1e-3 is clearly the better learning step as it converges before max iterations with the absolute value of loss being also similar to the loss value obtained at the 1000th step when learing rate is 1e-4

Confusion Matrix [1 points]

Predict labels on the test set and calculate the confusion matrix. Interpret the matrix to discuss true positives, true negatives, false positives, and false negatives.

```
# TODO
# YOUR CODE HERE

# Taking the values when Learning rate is 1e-2
```

```
(w, all_cost, iters) =
logistic_regression_gd(X_train,y_train,learning_rate = 0.001,max_iter
= 1000, tol=pow(10,-6))
y_pred = predict(X,w)
bin = (y_pred >= 0.0).astype(int)
tp = np.sum((y_test == 1) & (bin == 1))
tn = np.sum((y_test == 0) & (bin == 0))
fp = np.sum((y_test == 0) & (bin == 1))
fn = np.sum((y_test == 1) & (bin == 0))

# Print confusion matrix values
print("Confusion Matrix:")
print(f"True Positives (TP): {tp}")
print(f"True Negatives (TN): {tn}")
print(f"False Positives (FP): {fp}")
print(f"False Negatives (FN): {fn}")
```

```
Confusion Matrix:
True Positives (TP): 67
True Negatives (TN): 0
False Positives (FP): 138
False Negatives (FN): 0
```

Then calculate precision and recall manually.

- **Precision**:

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

- **Recall**:

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

```
# Calculate precision and recall
precision = tp / (tp + fp)# YOUR CODE HERE
recall = tp / (tp + fn)# YOUR CODE HERE

print(f"Precision: {precision:.2f}")
print(f"Recall: {recall:.2f}")
```

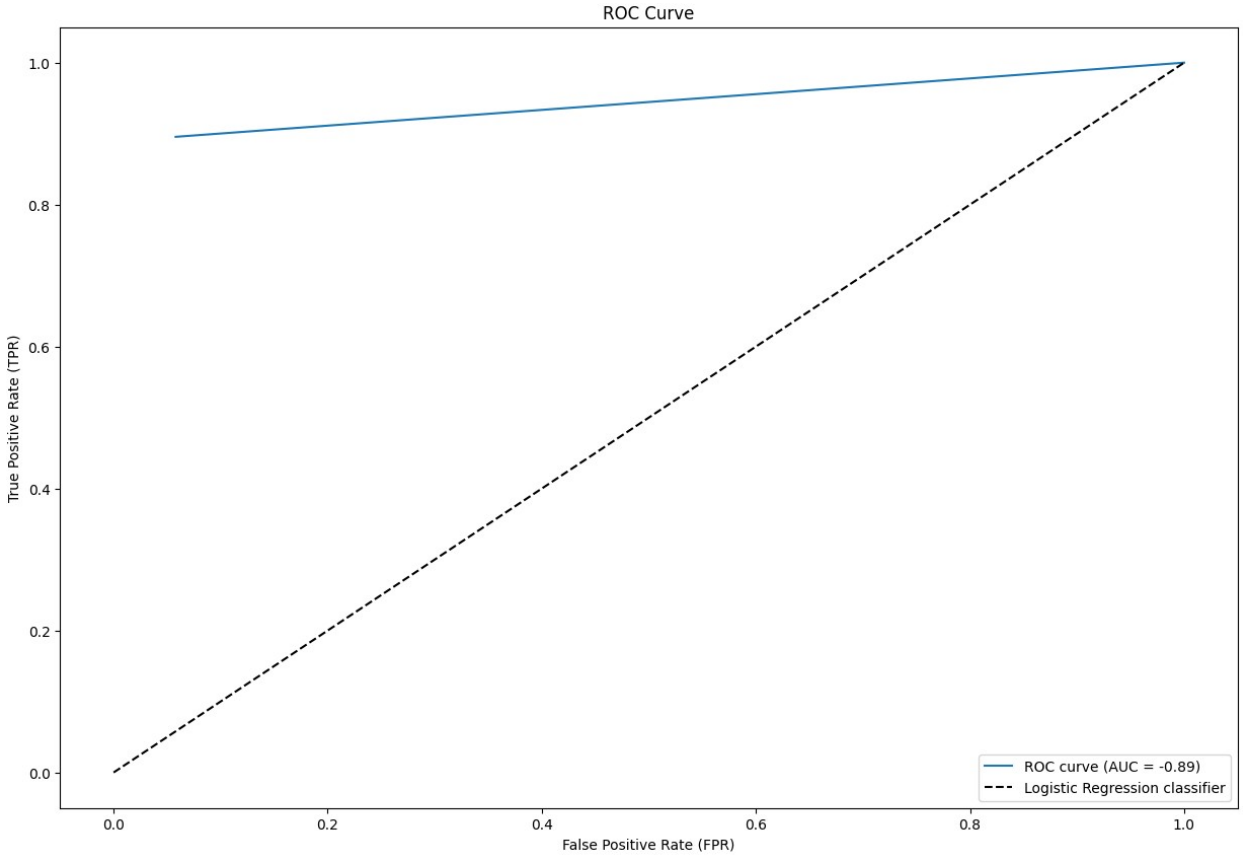
```
Precision: 0.33
Recall: 1.00
```

ROC& AUC [1 point]

After making predictions on the test set, calculate the true positive rate (TPR) and false positive rate (FPR) at various thresholds. Plot the ROC curve using TPR vs. FPR, then calculate the AUC, which represents the area under the ROC curve.

```python
# Function to calculate TPR and FPR for different thresholds
def calculate_roc(y_true, y_scores):
    # YOUR CODE HERE
    tpr = []
    fpr = []
    thresholds = np.linspace(0, 1, 201)

    for threshold in thresholds:
        bin = (y_scores >= threshold).astype(int)
        tp = np.sum((y_true == 1) & (bin == 1))
        tn = np.sum((y_true == 0) & (bin == 0))
        fp = np.sum((y_true == 0) & (bin == 1))
        fn = np.sum((y_true == 1) & (bin == 0))

        # Have to avoid zeros
        tpr.append(tp / (tp + fn) if (tp + fn) > 0 else 0)
        fpr.append(fp / (fp + tn) if (fp + tn) > 0 else 0)
    return tpr, fpr, thresholds

# Function to calculate AUC using the trapezoidal rule
def calculate_auc(fpr, tpr):
    # YOUR CODE HERE
    auc = np.trapz(tpr, fpr)
    return auc

# YOUR CODE HERE

(tpr, fpr, thresholds) = calculate_roc(y_test, y_pred)
plt.figure(figsize=(15, 10))
auc = calculate_auc(fpr, tpr)
plt.plot(fpr, tpr, label=f'ROC curve (AUC = {auc:.2f})')
plt.plot([0, 1], [0, 1], 'k--', label='Logistic Regression
classifier')
plt.xlabel('False Positive Rate (FPR)')
plt.ylabel('True Positive Rate (TPR)')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.show()
```

```
/var/folders/ht/nk3vbn95521gs7wwfq7vdngh0000gn/T/
ipykernel_82213/884671067.py:23: DeprecationWarning: `trapz` is
deprecated. Use `trapezoid` instead, or one of the numerical
integration functions in `scipy.integrate`.
  auc = np.trapz(tpr, fpr)
```


ROC Curve
True Positive Rate (TPR)
False Positive Rate (FPR)
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (AUC = -0.89)
Logistic Regression classifier