fall2022_hw2

October 27, 2022

1 CS171-EE142 - Fall 2022 - Homework 2

2 Due: Thursday, October 28, 2022 @ 11:59pm

2.0.1 Maximum points: 100 pts

2.1 Submit your solution to Gradescope:

1. Submit a single PDF to **HW2**
2. Submit your jupyter notebook to **HW2-code**

Notice: In Markdown, when you write in LaTeX math mode, do not leave any leading and trailing whitespaces inside the dollar signs (\$). For example, write `(dollarSign)\mathbf(dollarSign)(dollarSign)` instead of `(dollarSign)(space)\mathbf{w}(dollarSign)`. Otherwise, nbconvert will throw an error and the generated pdf will be incomplete. This is a bug of nbconvert.

See the additional submission instructions at the end of this notebook

2.2 Enter your information below:

2.2.1 Your Name (submitter): Yash Aggarwal

2.2.2 Your student ID (submitter): 862333037

By submitting this notebook, I assert that the work below is my own work, completed for this course. Except where explicitly cited, none of the portions of this notebook are duplicated from anyone else's work or my own previous work.

2.3 Academic Integrity

Each assignment should be done individually. You may discuss general approaches with other students in the class, and ask questions to the TAs, but you must only submit work that is yours . If you receive help by any external sources (other than the TA and the instructor), you must properly credit those sources, and if the help is significant, the appropriate grade reduction will be applied. If you fail to do so, the instructor and the TAs are obligated to take the appropriate

actions outlined at http://conduct.ucr.edu/policies/academicintegrity.html . Please read carefully the UCR academic integrity policies included in the link.

[1]:
```
# Standard library imports.
import random as rand

# Related third party imports.
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
import math

# Local application/library specific imports.
# import here if you write .py script
```

2.4 Question 1: Linear Regression [70 pts]

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.

2.4.1 Getting data

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

I have supplied a list `names` of the column headers. You will have to set the options in the `read_csv` command to correctly delimit the data in the file and name the columns correctly.

[2]:
```
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,delim_whitespace=True,names=names,na_values='?')

df
```

[2]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	\
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296.0	
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242.0	
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242.0	
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222.0	
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222.0	
..	…	…	…	…	…	…	…	…	…	…	
501	0.06263	0.0	11.93	0	0.573	6.593	69.1	2.4786	1	273.0	
502	0.04527	0.0	11.93	0	0.573	6.120	76.7	2.2875	1	273.0	
503	0.06076	0.0	11.93	0	0.573	6.976	91.0	2.1675	1	273.0	
504	0.10959	0.0	11.93	0	0.573	6.794	89.3	2.3889	1	273.0	
505	0.04741	0.0	11.93	0	0.573	6.030	80.8	2.5050	1	273.0	

	PTRATIO	B	LSTAT	PRICE
0	15.3	396.90	4.98	24.0
1	17.8	396.90	9.14	21.6
2	17.8	392.83	4.03	34.7
3	18.7	394.63	2.94	33.4
4	18.7	396.90	5.33	36.2
..	…	…	…	…
501	21.0	391.99	9.67	22.4
502	21.0	396.90	9.08	20.6
503	21.0	396.90	5.64	23.9
504	21.0	393.45	6.48	22.0
505	21.0	396.90	7.88	11.9

[506 rows x 14 columns]

2.4.2 Basic Manipulations on the Data

What is the shape of the data? How many attributes are there? How many samples? Print a statement of the form:

```
num samples=xxx, num attributes=yy
```

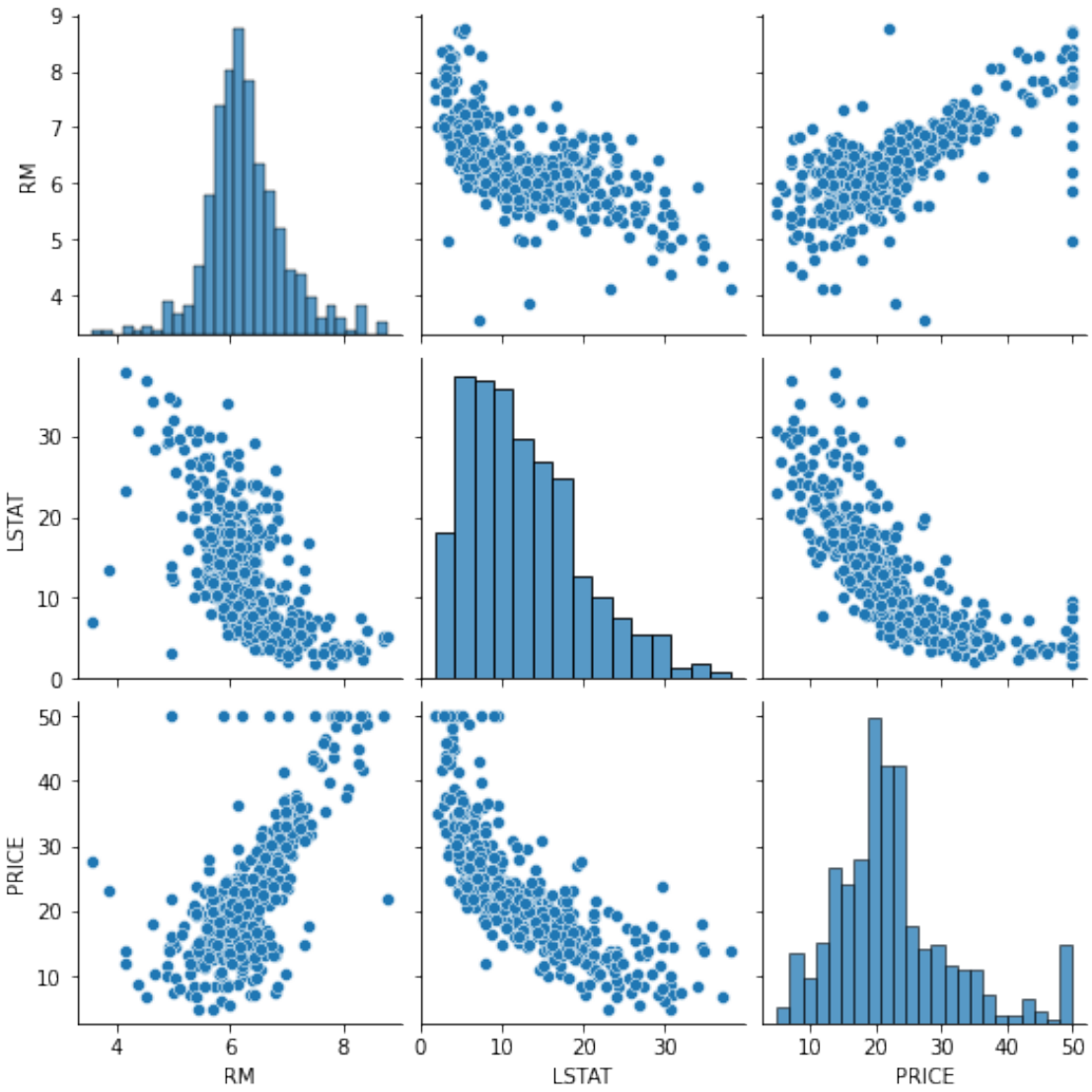
[3]:
```
data_samples, data_attributes = df.shape
print ('num of samples :', data_samples)
print ('num of attributes :', data_attributes)
```

```
num of samples : 506
num of attributes : 14
```

In order to properly test linear regression, we first need to find a set of correlated variables, so that we use one to predict the other. Consider the following scatterplots:

[4]:
```
# RM - average number of rooms per dwelling
# LSTAT - % lower status of the population

sns.pairplot(df[['RM','LSTAT','PRICE']])
```

[4]:
```
<seaborn.axisgrid.PairGrid at 0x28499110b50>
```



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

[5]:
```
# TODO
y = np.array(df['PRICE'])
print (type(y))
y
```

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

[5]:
```
array([24. , 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, 18.9, 15. ,
       18.9, 21.7, 20.4, 18.2, 19.9, 23.1, 17.5, 20.2, 18.2, 13.6, 19.6,
       15.2, 14.5, 15.6, 13.9, 16.6, 14.8, 18.4, 21. , 12.7, 14.5, 13.2,
       13.1, 13.5, 18.9, 20. , 21. , 24.7, 30.8, 34.9, 26.6, 25.3, 24.7,
       21.2, 19.3, 20. , 16.6, 14.4, 19.4, 19.7, 20.5, 25. , 23.4, 18.9,
       35.4, 24.7, 31.6, 23.3, 19.6, 18.7, 16. , 22.2, 25. , 33. , 23.5,
       19.4, 22. , 17.4, 20.9, 24.2, 21.7, 22.8, 23.4, 24.1, 21.4, 20. ,
       20.8, 21.2, 20.3, 28. , 23.9, 24.8, 22.9, 23.9, 26.6, 22.5, 22.2,
       23.6, 28.7, 22.6, 22. , 22.9, 25. , 20.6, 28.4, 21.4, 38.7, 43.8,
       33.2, 27.5, 26.5, 18.6, 19.3, 20.1, 19.5, 19.5, 20.4, 19.8, 19.4,
       21.7, 22.8, 18.8, 18.7, 18.5, 18.3, 21.2, 19.2, 20.4, 19.3, 22. ,
       20.3, 20.5, 17.3, 18.8, 21.4, 15.7, 16.2, 18. , 14.3, 19.2, 19.6,
       23. , 18.4, 15.6, 18.1, 17.4, 17.1, 13.3, 17.8, 14. , 14.4, 13.4,
       15.6, 11.8, 13.8, 15.6, 14.6, 17.8, 15.4, 21.5, 19.6, 15.3, 19.4,
       17. , 15.6, 13.1, 41.3, 24.3, 23.3, 27. , 50. , 50. , 50. , 22.7,
       25. , 50. , 23.8, 23.8, 22.3, 17.4, 19.1, 23.1, 23.6, 22.6, 29.4,
       23.2, 24.6, 29.9, 37.2, 39.8, 36.2, 37.9, 32.5, 26.4, 29.6, 50. ,
       32. , 29.8, 34.9, 37. , 30.5, 36.4, 31.1, 29.1, 50. , 33.3, 30.3,
       34.6, 34.9, 32.9, 24.1, 42.3, 48.5, 50. , 22.6, 24.4, 22.5, 24.4,
       20. , 21.7, 19.3, 22.4, 28.1, 23.7, 25. , 23.3, 28.7, 21.5, 23. ,
       26.7, 21.7, 27.5, 30.1, 44.8, 50. , 37.6, 31.6, 46.7, 31.5, 24.3,
       31.7, 41.7, 48.3, 29. , 24. , 25.1, 31.5, 23.7, 23.3, 22. , 20.1,
       22.2, 23.7, 17.6, 18.5, 24.3, 20.5, 24.5, 26.2, 24.4, 24.8, 29.6,
       42.8, 21.9, 20.9, 44. , 50. , 36. , 30.1, 33.8, 43.1, 48.8, 31. ,
       36.5, 22.8, 30.7, 50. , 43.5, 20.7, 21.1, 25.2, 24.4, 35.2, 32.4,
       32. , 33.2, 33.1, 29.1, 35.1, 45.4, 35.4, 46. , 50. , 32.2, 22. ,
       20.1, 23.2, 22.3, 24.8, 28.5, 37.3, 27.9, 23.9, 21.7, 28.6, 27.1,
       20.3, 22.5, 29. , 24.8, 22. , 26.4, 33.1, 36.1, 28.4, 33.4, 28.2,
       22.8, 20.3, 16.1, 22.1, 19.4, 21.6, 23.8, 16.2, 17.8, 19.8, 23.1,
       21. , 23.8, 23.1, 20.4, 18.5, 25. , 24.6, 23. , 22.2, 19.3, 22.6,
       19.8, 17.1, 19.4, 22.2, 20.7, 21.1, 19.5, 18.5, 20.6, 19. , 18.7,
       32.7, 16.5, 23.9, 31.2, 17.5, 17.2, 23.1, 24.5, 26.6, 22.9, 24.1,
       18.6, 30.1, 18.2, 20.6, 17.8, 21.7, 22.7, 22.6, 25. , 19.9, 20.8,
       16.8, 21.9, 27.5, 21.9, 23.1, 50. , 50. , 50. , 50. , 50. , 13.8,
       13.8, 15. , 13.9, 13.3, 13.1, 10.2, 10.4, 10.9, 11.3, 12.3,  8.8,
        7.2, 10.5,  7.4, 10.2, 11.5, 15.1, 23.2,  9.7, 13.8, 12.7, 13.1,
       12.5,  8.5,  5. ,  6.3,  5.6,  7.2, 12.1,  8.3,  8.5,  5. , 11.9,
       27.9, 17.2, 27.5, 15. , 17.2, 17.9, 16.3,  7. ,  7.2,  7.5, 10.4,
        8.8,  8.4, 16.7, 14.2, 20.8, 13.4, 11.7,  8.3, 10.2, 10.9, 11. ,
        9.5, 14.5, 14.1, 16.1, 14.3, 11.7, 13.4,  9.6,  8.7,  8.4, 12.8,
```

```
10.5, 17.1, 18.4, 15.4, 10.8, 11.8, 14.9, 12.6, 14.1, 13. , 13.4,
15.2, 16.1, 17.8, 14.9, 14.1, 12.7, 13.5, 14.9, 20. , 16.4, 17.7,
19.5, 20.2, 21.4, 19.9, 19. , 19.1, 19.1, 20.1, 19.9, 19.6, 23.2,
29.8, 13.8, 13.3, 16.7, 12. , 14.6, 21.4, 23. , 23.7, 25. , 21.8,
20.6, 21.2, 19.1, 20.6, 15.2,  7. ,  8.1, 13.6, 20.1, 21.8, 24.5,
23.1, 19.7, 18.3, 21.2, 17.5, 16.8, 22.4, 20.6, 23.9, 22. , 11.9])
```

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:

```
The mean house price is xx.yy thousands of dollars.
Only x.y percent are above $40k.
```

[6]:
```
# TODO
print ('The mean house price is {00:.2f} thousands of dollars.'.format(y.
 ↪mean()))

greater_than_40k_list = [i for i in y if i > 40 ]

print ('Only {0:.1f} percent are above $40k'.format(␣
 ↪(len(greater_than_40k_list)/len(y))*100 ))
```

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

2.4.3 Visualizing the Data

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.

[7]:
```
import matplotlib
import matplotlib.pyplot as plt
%matplotlib inline
```

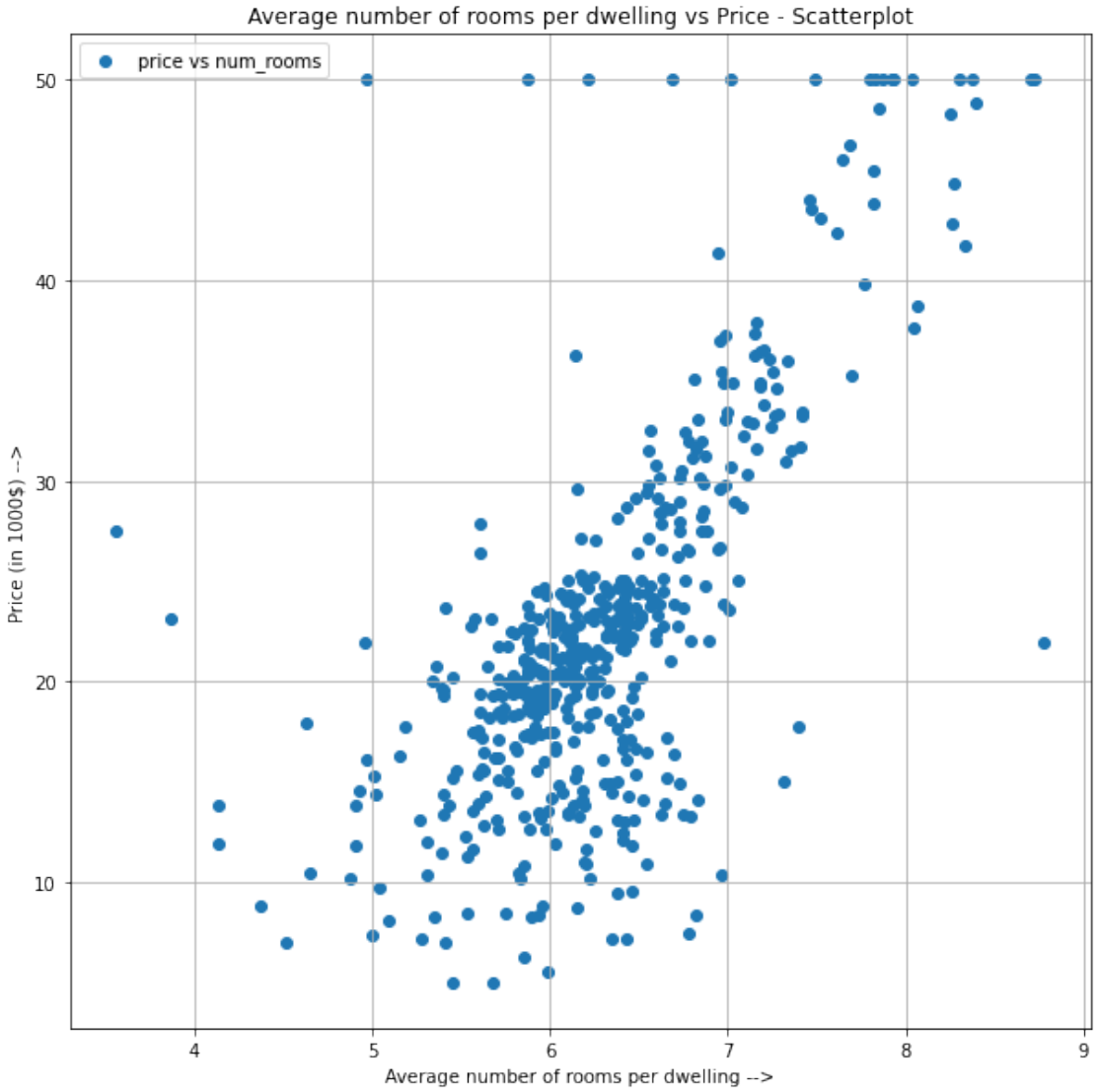
Similar to the `y` vector, create a predictor vector `x` containing the values in the `RM` column, which represents the average number of rooms in each region.

[8]:
```
# TODO
x = np.array(df['RM'])
x.shape
```

[8]:
```
(506,)
```

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.

[10]:
```
# TODO
fig = plt.figure(figsize=(10,10))
plt.scatter(x, y, label="price vs num_rooms")
plt.xlabel('Average number of rooms per dwelling -->')
plt.ylabel('Price (in 1000$) -->')
plt.title('Average number of rooms per dwelling vs Price - Scatterplot')
plt.legend(loc='upper left')
plt.grid()
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.

2.4.4 Question 1a. Derivation of a simple linear model for a single feature [10 pts]

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

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

$$y = wx + b,$$

where $w \in \mathbb{R}$ and $b \in \mathbb{R}$.

In the class, we looked at the derivation of optimal value of w when $b = 0$. The squared loss function can be written as

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

and the optimal value of $w*$ that minimizes $L(w)$ can be written as

$$w^* = (\sum_{i=1}^{N} x_i^2)^{-1} (\sum_{i=1}^{N} x_i y_i)$$

.

Now let us include b in our model. Show that the optimal values of w^*, b^* that minimize the loss function

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

can be written as

$$w^* = (\sum_{i} (x_i - \bar{x})^2)^{-1} (\sum_{i} (x_i - \bar{x})(y_i - \bar{y}))$$

and

$$b^* = \bar{y} - w^* \bar{x},$$

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

TODO: Your derivation goes here.

- *Hint. Set the partial derivative of $L(w, b)$ with respect to w and b to zero.*
- Type using latex commands and explain your steps

The squared loss function for each datapoint of the entire dataset is computed as

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

b can can be calculated as partial derivative of Loss

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

for optimal b, this derivative should be equal to 0

$$b^* = \frac{\partial L(w, b)}{\partial b} = \sum_{i=1}^{N} (wx_i + b - y_i)^2 = 0$$

taking partial derivative w.r.t b and using chain rule and product rule

$$=> \sum_{i=1}^{N} 2(wx_i + b - y_i)(1) = 0$$

multiplying the constant 2 and sumation

$$=> 2\sum_{i=1}^{N} wx_i + 2\sum_{i=1}^{N} b - 2\sum_{i=1}^{N} y_i = 0$$

dividing L.H.S and R.H.S by 2 and rearranging terms

$$=> w\sum_{i=1}^{N} x_i + Nb = \sum_{i=1}^{N} y_i$$

on solving for b. we get

$$=> Nb = \sum_{i=1}^{N} y_i - w\sum_{i=1}^{N} x_i$$

divide by N

$$=> b = (\sum_{i=1}^{N} y_i)/N - w(\sum_{i=1}^{N} x_i)/N$$

where

$$(\sum_{i=1}^{N} y_i)/N) = Y_{mean}$$

or

$$\bar{y} = \frac{1}{N}\sum_{i} y_i$$

and

$$(\sum_{i=1}^{N} x_i)/N) = X_{mean}$$

or

$$\bar{x} = \frac{1}{N}\sum_{i} x_i$$

Therefore,

$$b^* = Y_{mean} - wX_{mean}$$

Similarly, w can can be calculated as partial derivative of Loss

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

for optimal w, this derivative should be equal to 0

$$w^* = \frac{\partial L(w,b)}{\partial b} = \sum_{i=1}^{N}(wx_i + b - y_i)^2 = 0$$

taking partial derivative w.r.t w and using chain rule and product rule

$$=> \sum_{i=1}^{N} 2(wx_i + b - y_i)(x_i) = 0$$

dividing L.H.S and R.H.S by 2

$$=> \sum_{i=1}^{N}(wx_i^2 + bx_i - y_i x_i) = 0$$

substituting following b in above equation

$$b = \bar{y} - w\bar{x}$$

we get

$$=> \sum_{i=1}^{N}(wx_i^2 + (\bar{y} - w\bar{x})x_i - y_i x_i) = 0$$

$$=> \sum_{i=1}^{N}(wx_i^2 + \bar{y}x_i - w\bar{x}x_i - y_i x_i) = 0$$

On re-arranging terms

$$=> \sum_{i=1}^{N}(w[x_i^2 - \bar{x}x_i] + [\bar{y}x_i - y_i x_i]) = 0$$

$$=> w\sum_{i=1}^{N}([x_i^2 - \bar{x}x_i]) = \sum_{i=1}^{N}([y_i x_i - \bar{y}x_i])$$

On solving for w, we get

$$=> w = \frac{\sum_{i=1}^{N}(y_i x_i - \bar{y}x_i)}{\sum_{i=1}^{N}(x_i^2 - \bar{x}x_i)}$$

or this can be written as

$$w^* = \frac{\sum_{i=1}^{N}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

2.4.5 Question 1b. Fitting a linear model using a single feature [10 pts]

Next we will write a function to perform a linear fit. Use the formulae above to compute the parameters w, b in the linear model $y = wx + b$.

[11]:
```
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 complete the code below
    w = 0
    b = 0

    x_bar = sum(x)/len(x)
    y_bar = sum(y)/len(y)

    denominator = 0
    numerator = 0

    for x_val,y_val in zip(x,y):
        diff_denom_x = (x_val - x_bar) ** 2
        denominator += diff_denom_x

        diff_numer_x = (x_val - x_bar)
        diff_numer_y = (y_val - y_bar)
        numerator += (diff_numer_x * diff_numer_y)


    w = numerator / denominator

    b = y_bar - (w*x_bar)


    return w, b
```

Using the function `fit_linear` above, print the values `w`, `b` for the linear model of price vs. number of rooms.

[12]:
```
# TODO
w, b = fit_linear(x,y)
print('w = {0:5.1f}, b = {1:5.1f}'.format(w,b))
```

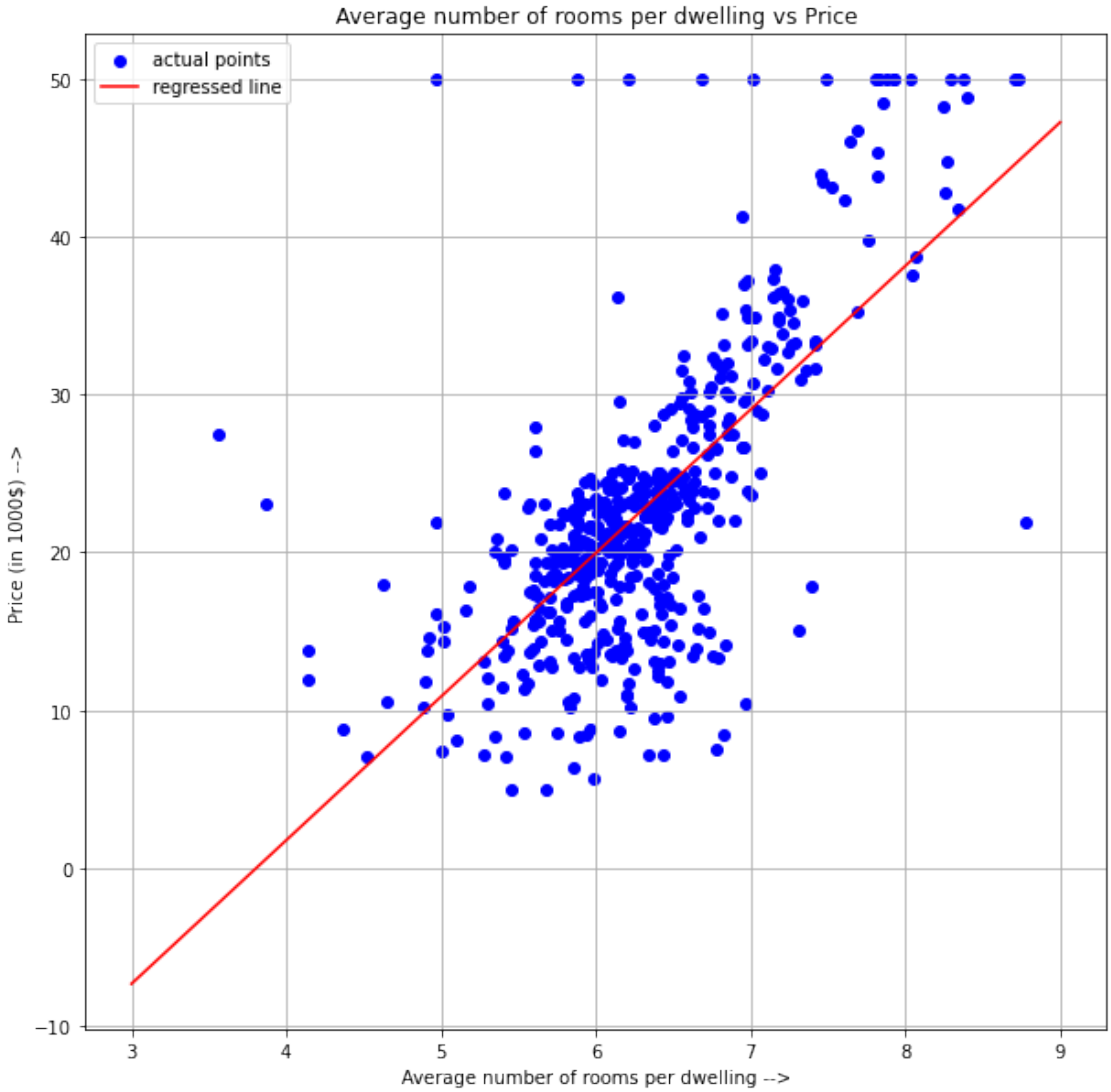
```
w =   9.1, b = -34.7
```

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

- As the w is positive, the price should increase with the increase in number of rooms

Replot the scatter plot above, but now with the regression line. You can create the regression line by creating points `xp` from say min(x) to max(x), computing the linear predicted values `yp` on those points and plotting `yp` vs. `xp` on top of the above plot.

[13]:
```
# TODO
# Points on the regression line

xp = [i for i in range(math.floor(min(x)), math.floor(max(x)) + 2)]
yp = [w*i + b for i in xp]


# Scatterplot
fig = plt.figure(figsize=(10,10))
plt.scatter(x, y,c='b',label = 'actual points')
plt.xlabel('Average number of rooms per dwelling -->')
plt.ylabel('Price (in 1000$) -->')
plt.title('Average number of rooms per dwelling vs Price')
plt.plot(xp,yp,'r',label="regressed line")
plt.legend(loc='upper left')
# Regression line
plt.grid()
plt.show()
```



2.4.6 Question 1c. Linear regression with multiple features/attributes [20 pts]

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

In this exercise, the target variable will 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.

[15]:
```
xnames = names[:-1]
print(names[:-1])
```

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

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'])
```

[16]:
```
# TODO
X = np.array(df[['CRIM', 'RM', 'LSTAT']])
Y = np.array(df['PRICE'])

print (X[:5],Y[:5])
```

```
[[6.320e-03 6.575e+00 4.980e+00]
 [2.731e-02 6.421e+00 9.140e+00]
 [2.729e-02 7.185e+00 4.030e+00]
 [3.237e-02 6.998e+00 2.940e+00]
 [6.905e-02 7.147e+00 5.330e+00]] [24.  21.6 34.7 33.4 36.2]
```

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 directly.

Linear regression by solving least-squares problem (direct solution)

Suppose we have N pairs of training samples $(x_1, y_1), \ldots, (x_N, y_N)$, where $\mathbf{x}_i \in \mathbb{R}^d$ and $y_i \in \mathbb{R}$.

We want to perform a linear fit over all the data features as

$$y = \mathbf{w}^T \mathbf{x} + b,$$

where $\mathbf{w} \in \mathbb{R}^d$ and $b \in \mathbb{R}$.

We saw in the class that we can write all the training data as a linear system

$$\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} - & \mathbf{x}_1^T & - \\ & \vdots & \\ - & \mathbf{x}_N^T & - \end{bmatrix} \mathbf{w} + b,$$

which can be written as

$$\begin{bmatrix} y_1 \\ \vdots \\ y_N \end{bmatrix} = \begin{bmatrix} 1 & \mathbf{x}_1^T \\ \vdots & \vdots \\ 1 & \mathbf{x}_N^T \end{bmatrix} \begin{bmatrix} b \\ \mathbf{w} \end{bmatrix}.$$

Let us write this system of linear equations in a compact form as

$$\mathbf{y} = \mathbf{X}\mathbf{w}, \tag{1}$$

where $\mathbf{X}$ is an $N \times d + 1$ matrix whose first column is all ones and $\mathbf{w}$ is a vector of length $d + 1$ whose first term is the constant and rest of them are the coefficients of the linear model.

The least-squares problem for the system above can be written as

$$\text{minimize } \frac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$$

for which the closed form solution can be written as

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}.$$

Append ones to the data matrix

To compute the coefficients $\mathbf{w}$, we first append a vector of ones to the data matrix. This can be performed with the `ones` command and `hstack`. Note that after we do this, `X` will have one more column than before.

[17]:
```
# TODO
# X before appending column of 1s
print (X[:5])
ones = np.ones((X.shape[0], 1))
X = np.hstack((X, ones))

# # X after appending column of 1s
print (X[:5])
```

```
[[6.320e-03 6.575e+00 4.980e+00]
 [2.731e-02 6.421e+00 9.140e+00]
 [2.729e-02 7.185e+00 4.030e+00]
 [3.237e-02 6.998e+00 2.940e+00]
 [6.905e-02 7.147e+00 5.330e+00]]
[[6.320e-03 6.575e+00 4.980e+00 1.000e+00]
```

```
[2.731e-02 6.421e+00 9.140e+00 1.000e+00]
[2.729e-02 7.185e+00 4.030e+00 1.000e+00]
[3.237e-02 6.998e+00 2.940e+00 1.000e+00]
[6.905e-02 7.147e+00 5.330e+00 1.000e+00]]
```

Split the Data into Training and Test

Split the data into training and test. Use 30% for test and 70% for training. 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`.

[18]:
```
from sklearn.model_selection import train_test_split

# TODO
(X_train, X_test, Y_train, Y_test) = train_test_split(X,Y,test_size=0.3)

print (X_train.shape, Y_train.shape)
print (X_test.shape, Y_test.shape)
```

```
(354, 4) (354,)
(152, 4) (152,)
```

Now let us compute the coefficients $\mathbf{w}$ using `Xtr,ytr` via the direct matrix inverse:

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}.$$

You may use `np.linalg.inv` to compute the inverse. For a small problem like this, it makes no difference. But, in general, using a matrix inverse like this is *much* slower computationally than using functions such as `lstsq` method or the `LinearRegression` class. In real world, you will never solve a least squares problem like this.

[19]:
```
# TODO
# compute w using the direct solution equation
w = (np.linalg.inv(X_train.T @ X_train) @ X_train.T) @ Y_train
w
```

[19]:
```
array([-0.09295765,  5.37946329, -0.50284828, -4.68298999])
```

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

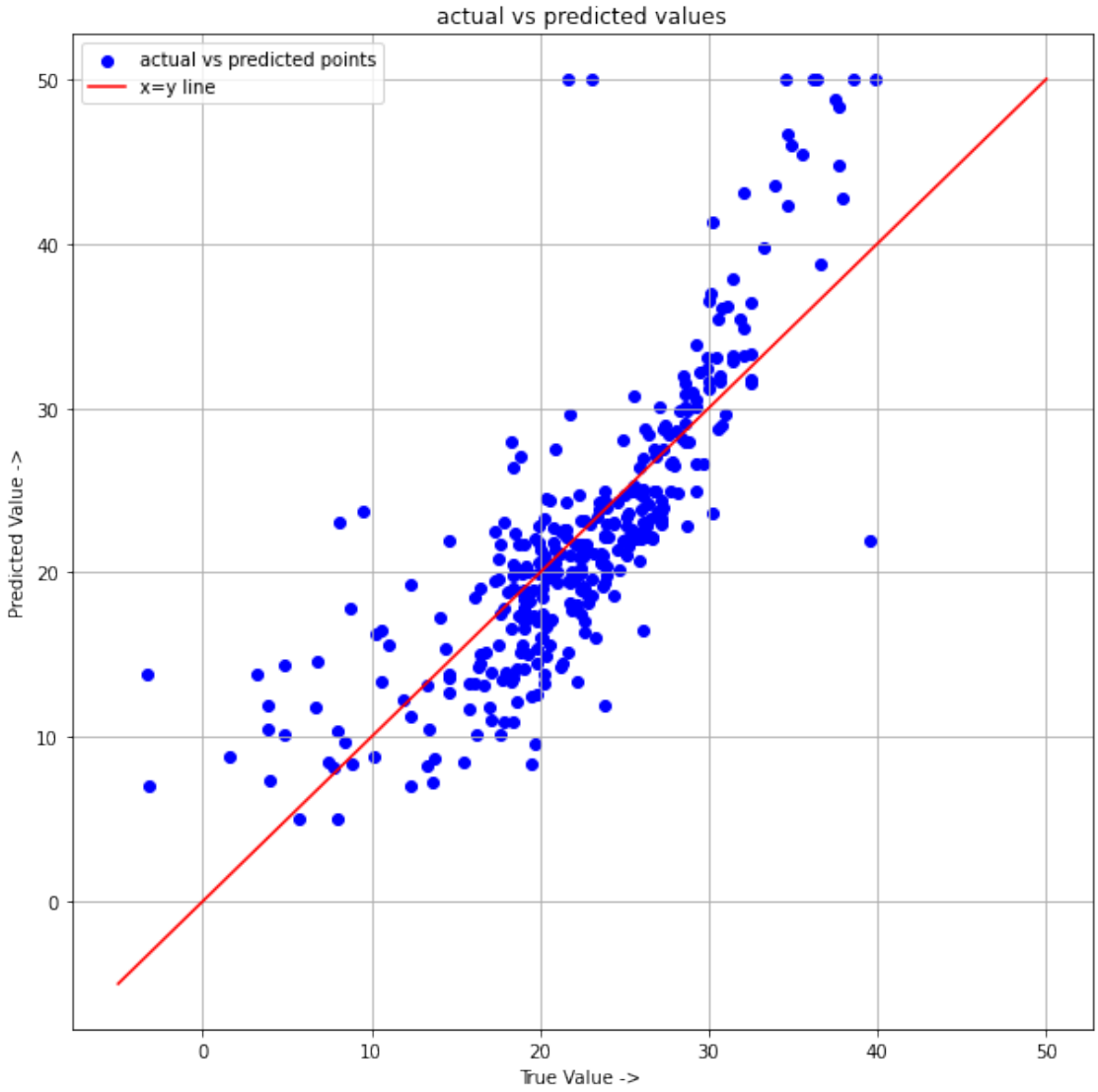
[20]:
```
# TODO
# your code here
yhat_tr = [w.T @ x for x in X_train]
yhat_tr

tot_loss = 0
for y_true, y_pred in zip(Y_train,yhat_tr):
    loss = (y_pred - y_true ) **2
    tot_loss += loss
```

```
print ('avg sq loss with computed weights= ', tot_loss / len(Y_train))

# Getting the loss using in-built functions for comparison
from sklearn import linear_model
regr = linear_model.LinearRegression()
regr.fit(X_train,Y_train)

y_pred = regr.predict(X_train)
print ('avg sq loss with sklearn implementation function= ', (np.sum(np.
 ↪square(y_pred - Y_train)))/len(Y_train))
```

```
avg sq loss with computed weights=  26.31362289367486
avg sq loss with sklearn implementation function=  26.31362289367483
```

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

[21]:
```
# TODO

y_tr = [i for i in Y_train]
yh_tr = [i for i in yhat_tr]

x_line = np.linspace(-5,50,100)
y_line = x_line

fig = plt.figure(figsize = (10,10))
red = plt.scatter(yh_tr, y_tr, c='b', label='actual vs predicted points')
plt.plot(x_line,y_line,'r',label='x=y line')
plt.xlabel('True Value -> ')
plt.ylabel('Predicted Value ->')
plt.legend(loc='upper left')
plt.title('actual vs predicted values')
plt.grid()
plt.show()
```



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

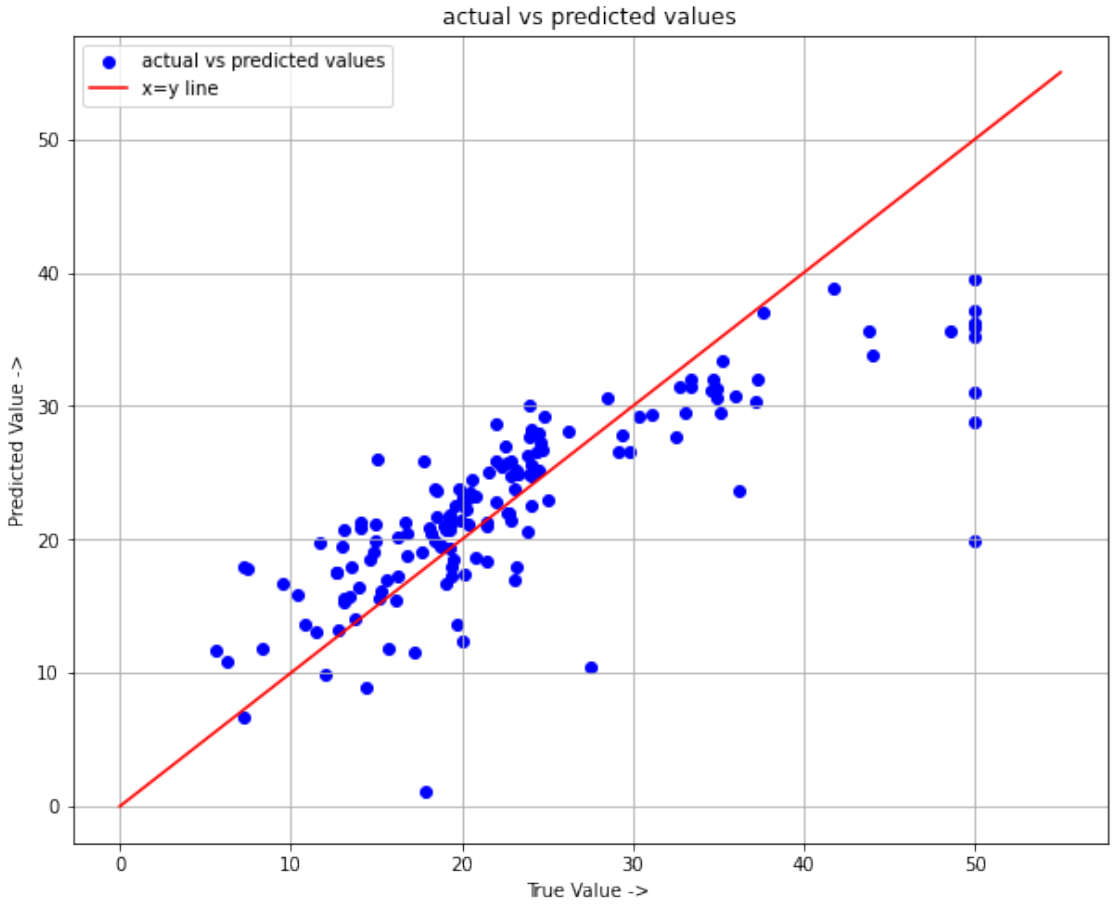
[22]:
```
# TODO
yhat_ts = [w.T @ x for x in X_test]
yhat_ts

tot_loss = 0
for y_true, y_pred in zip(Y_test,yhat_ts):
    loss = (y_pred - y_true ) **2
    tot_loss += loss
```

```
print ('avg sq loss on training set with computed weights= ', tot_loss /␣
 ↪len(Y_test))

# Getting the loss using in-built functions for comparison
from sklearn import linear_model
regr = linear_model.LinearRegression()
regr.fit(X_test,Y_test)

y_pred = regr.predict(X_test)
print ('avg sq loss on training set with sklearn implementation= ', (np.sum(np.
 ↪square(y_pred - Y_test)))/len(Y_test))
```

```
avg sq loss on training set with computed weights=  39.19185165086407
avg sq loss on training set with sklearn implementation=  36.079809069406636
```

Create a scatter plot of the actual vs. predicted values of y on the test data.

[23]:
```
# TODO

x_cor = [i for i in range(len(Y_test))]
y_ts = [i for i in Y_test]
yh_ts = [i for i in yhat_ts]


x_line = np.linspace(0,55,100)
y_line = x_line

fig = plt.figure(figsize = (10,8))
blue = plt.scatter(y_ts, yh_ts, c='b',label="actual vs predicted values")
red = plt.plot(x_line,y_line,'r',label="x=y line")
plt.xlabel('True Value -> ')
plt.ylabel('Predicted Value ->')
plt.legend(loc='upper left')
plt.title('actual vs predicted values')
plt.grid()
plt.show()
```



2.4.7 Question 1d: Gradient descent for linear regression [20 pts]

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)):
```

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 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 $L(\mathbf{w})$ 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`

Gradient can be computed as

$$\nabla_{\mathbf{w}} L = \mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}).$$

Estimate will be updated as $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} L$ at every iteration.

Note that the w in this derivation includes the constant term and X is a matrix that has ones column appended to it.

[24]:
```
# TODO
# Implement gradient descent for linear regression

def compute_cost(X,w,y):
    L = (np.linalg.norm(y-(X@w),2))**2
    return L

def compute_gradient(X,w,y):
    L = np.matmul(X.T,(np.matmul(X, w) - y))
    return L

def linear_regression_gd(X,y,learning_rate = 0.
 ↪00001,max_iter=10000,tol=pow(10,-5)):

    iters = 0
    all_cost = []
    w = np.zeros(X[0].shape[0])

    for epoch in range(max_iter):

        grad = compute_gradient(X,w,y)
        cost = compute_cost(X,w,y)
        all_cost.append(cost)
        w = w - learning_rate*grad

        iters += 1

        check_err = np.absolute((all_cost[epoch] - all_cost[epoch-1])/
 ↪all_cost[epoch-1])
        if epoch > 1 and check_err <= tol:
            print ('break by tolerence', check_err)
            print ('iter', iters)
            break
```

```
    return w, all_cost, iters
```

2.4.8 Question 1e: Convergence plots [10 pts]

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 $L(\mathbf{w})$ as a function of the number of iterations, and ideally, what we would like to see is $L(\mathbf{w})$ 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 vs. iterations) [5]
- What do you observe? [5]

Important: In reality, when we are running gradient descent, we should be checking convergence based on the validation error (i.e., we would have to split our training set into e.g., 70/30 training'/validation subsets, use the new training set to calculate the gradient descent updates and evaluate the error both on the training set and the validation set, and as soon as the validation loss stops improving, we stop training. In order to keep things simple, in this assignment we are only looking at the training loss, but as long as you have a function

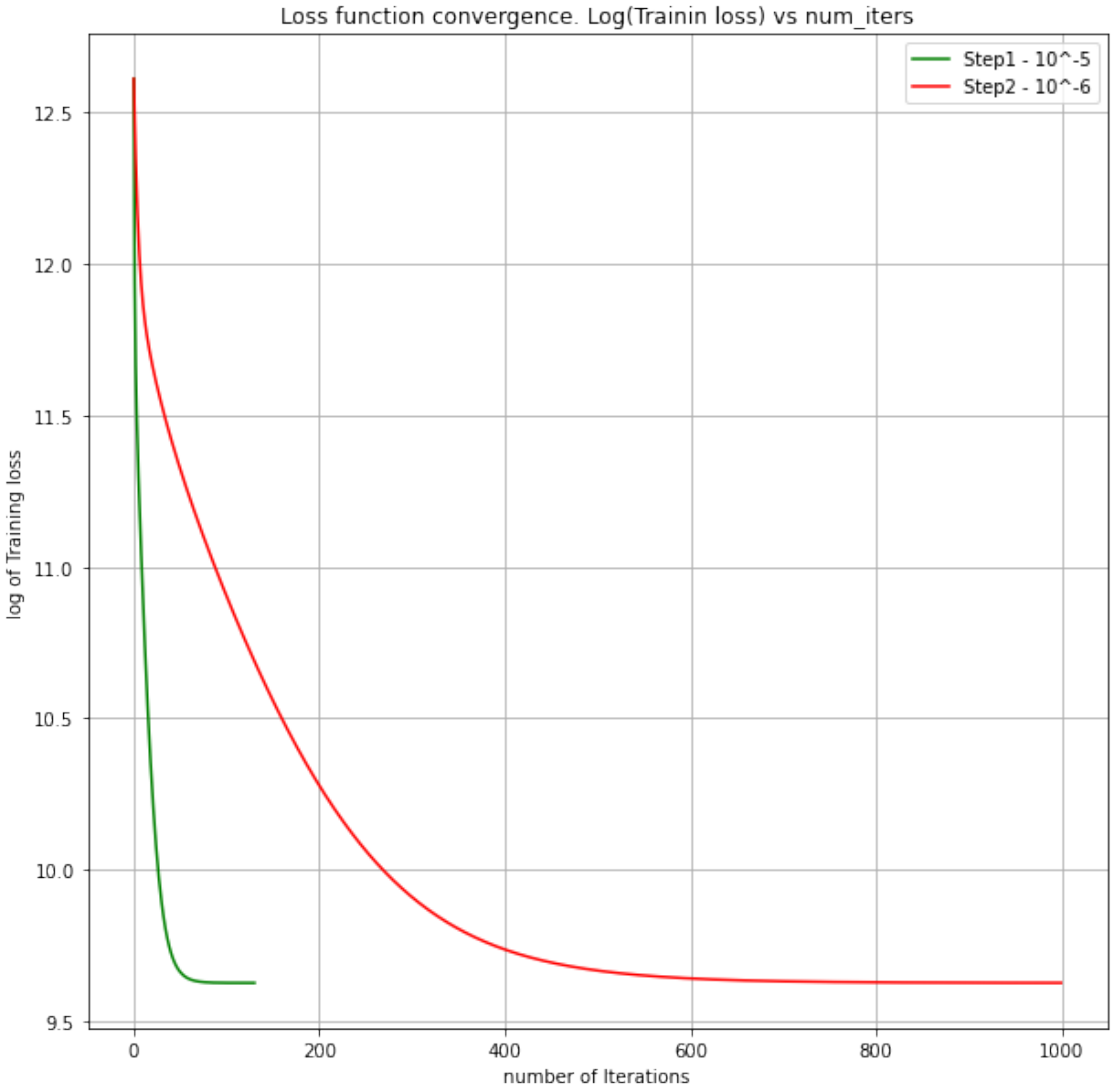
```
def compute_cost(X,w,y):
```

that calculates the loss for a given X, y, and set of parameters you have, you can always compute it on the validation portion of X and y (that are not used for the updates).

[25]:
```
# TODO
# test gradient descent with step size 0.00001
# test gradient descent with step size 0.000001
fig = plt.figure(0, figsize=(10,10))

(w1, all_cost1,iters1) = linear_regression_gd(X,Y,learning_rate = 0.
 ↪00001,max_iter = 1000, tol=pow(10,-6))
green, = plt.plot([i for i in range(iters1)], [math.log(i) for i in all_cost1[0:
 ↪iters1]], 'green')


(w2, all_cost2,iters2) = linear_regression_gd(X,Y,learning_rate = 0.
 ↪000001,max_iter = 1000, tol=pow(10,-6))
red, = plt.plot([i for i in range(iters2)],[math.log(i) for i in all_cost2[0:
 ↪iters2]],'red')
```

```
plt.legend([green, red], ['Step1 - 10^-5', 'Step2 - 10^-6'])
plt.xlabel('number of Iterations')
plt.ylabel('log of Training loss')
plt.title('Loss function convergence. Log(Trainin loss) vs num_iters')
plt.grid()
plt.show()
```

```
break by tolerence 9.861803124768551e-07
iter 131
```



Observations:

1. The Gradient Descent(GD) with lower step size, red in this case, takes more iterations to

come to same result than GD with larger step size, Green in this case.
2. Red has still not converged and is limited by number of iterations whilst green is limited by our tolerance value.

2.4.9 Question 2. Logistic regression [30 pts]

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/ml/machine-learning-databases/breast-cancer-wisconsin

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.

2.4.10 Loading and visualizing the Breast Cancer Data

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

[26]:
```
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)
```

[26]:

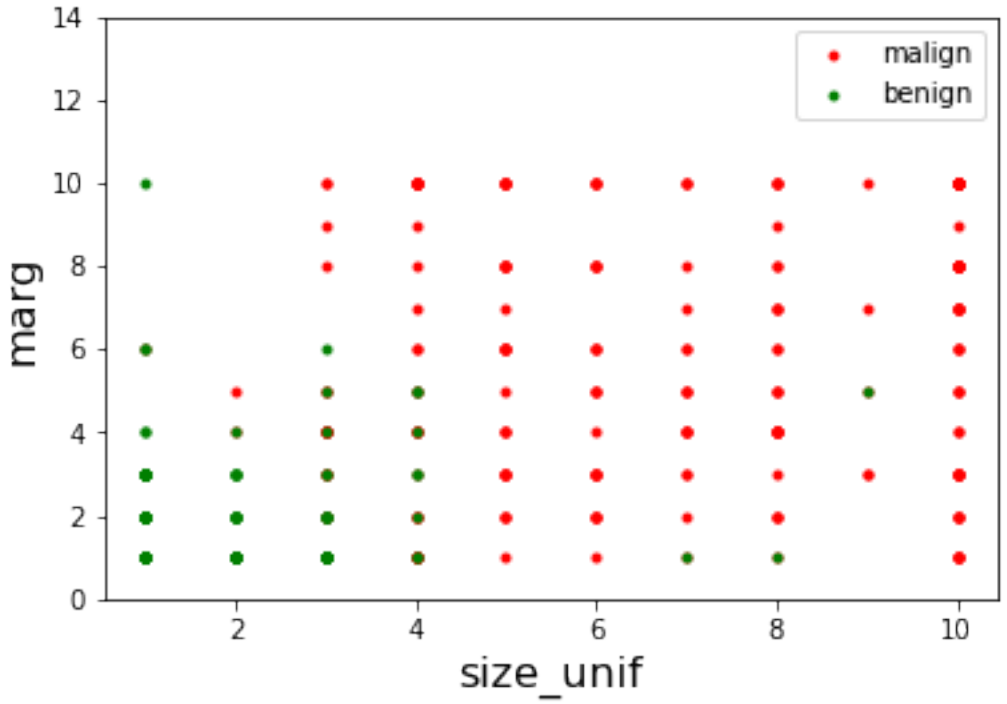
	id	thick	size_unif	shape_unif	marg	cell_size	bare	chrom	\
0	1000025	5	1	1	1	2	1.0	3	
1	1002945	5	4	4	5	7	10.0	3	
2	1015425	3	1	1	1	2	2.0	3	
3	1016277	6	8	8	1	3	4.0	3	
4	1017023	4	1	1	3	2	1.0	3	
5	1017122	8	10	10	8	7	10.0	9	

	normal	mit	class
0	1	1	2
1	2	1	2
2	1	1	2
3	7	1	2
4	1	1	2
5	7	1	4

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.

[27]:
```
# 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')
```

[27]:
```
<matplotlib.legend.Legend at 0x2849e560c40>
```



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.

2.4.11 Logistic function

We will build a binary classifier using *logistic regression*. In logistic regression, we do not just output an estimate of the class label. Instead, we ouput a *probability*, an estimate of how likely the sample is one class or the other. That is our output is a number from 0 to 1 representing the likelihood:

$$P(y = 1|x)$$

which is our estimate of the probability that the sample is one class (in this case, a malignant sample) based on the features in `x`. This is sometimes called a *soft classifier*.

In logistic regression, we assume that likelihood is of the form

$$P(y = 1|x) = \sigma(z), \quad z = w(1)x(1) + \cdots + w(d)x(d) + b = \mathbf{w}^T\mathbf{x} + b,$$

where $w(1), \ldots, w(d), b$ are the classifier weights and $\sigma(z)$ is the so-called *logistic* function:

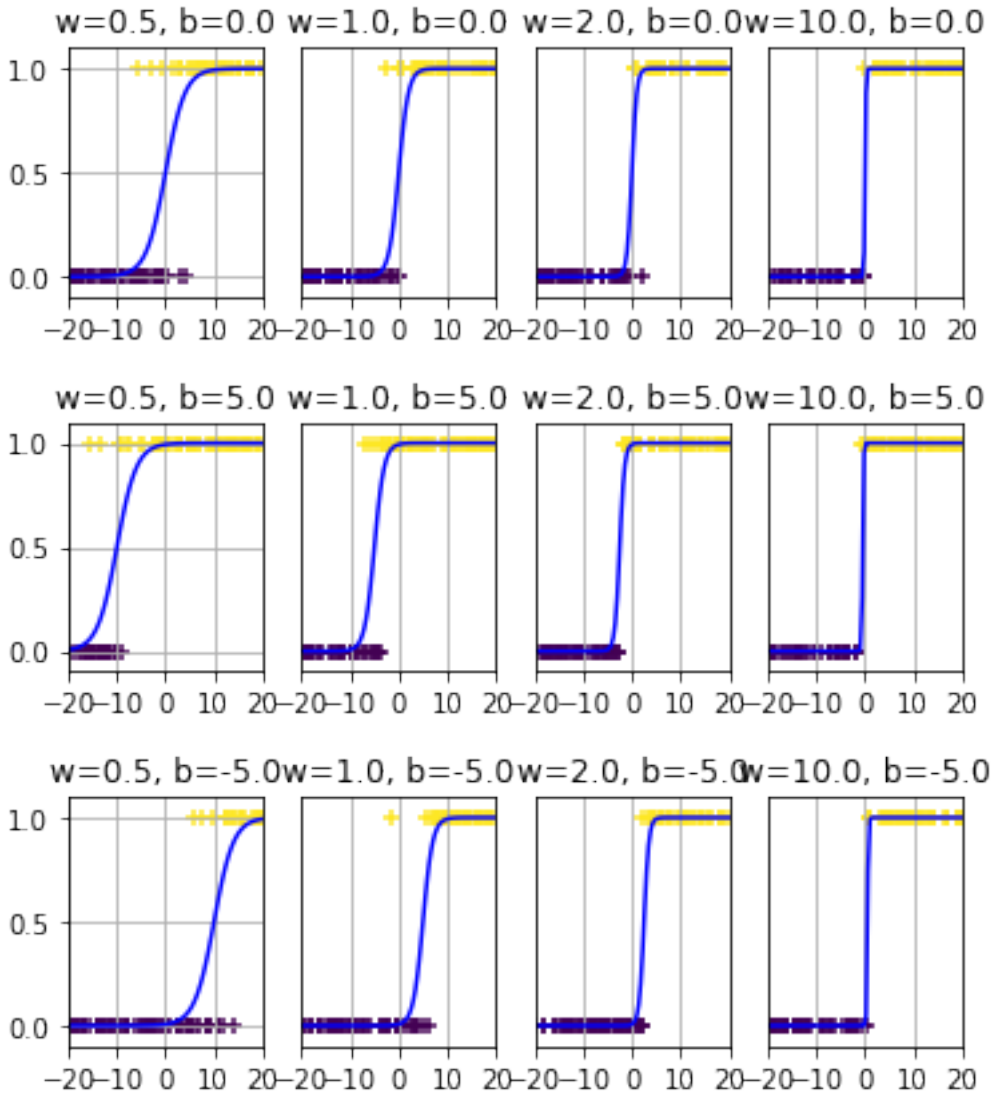
$$\sigma(z) = \frac{1}{1 + e^{-z}}.$$

To understand the logistic function, suppose x is a scalar and samples y are drawn with $P(y = 1|x) = f(wx + b)$ for some w and b. We plot these samples for different w, b.

[28]:
```
N = 100
xm = 20
ws = np.array([0.5,1,2,10])
bs = np.array([0, 5, -5])
wplot = ws.size
bplot = bs.size
iplot = 0
for b in bs:
  for w in ws:
    iplot += 1
    x  = np.random.uniform(-xm,xm,N)

    py = 1/(1+np.exp(-w*x-b))

    yp = np.array(np.random.rand(N) < py) # hard label for random points
    xp = np.linspace(-xm,xm,100)
    pyp = 1/(1+np.exp(-w*xp-b)) # soft label (probability) for the points

    plt.subplot(bplot,wplot,iplot)

    plt.scatter(x,yp,c=yp,edgecolors='none',marker='+')
    plt.plot(xp,pyp,'b-')
    plt.axis([-xm,xm,-0.1,1.1])
```

```
    plt.grid()
    if ((iplot%4)!=1):
        plt.yticks([])
    plt.xticks([-20,-10,0,10,20])
    plt.title('w={0:.1f}, b={1:.1f}'.format(w,b))

    plt.subplots_adjust(top=1.5, bottom=0.2, hspace=0.5, wspace=0.2)
```

```
C:\Users\yg375\AppData\Local\Temp\ipykernel_38020\4188138851.py:21: UserWarning:
You passed a edgecolor/edgecolors ('none') for an unfilled marker ('+').
Matplotlib is ignoring the edgecolor in favor of the facecolor.  This behavior
may change in the future.
  plt.scatter(x,yp,c=yp,edgecolors='none',marker='+')
```



We see that $\sigma(wx+b)$ represents the probability that $y=1$. The function $\sigma(wx) > 0.5$ 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.

2.4.12 Fitting the Logistic Model on Two Variables

We will fit the logistic model on the two variables `size_unif` and `marg` that we were looking at earlier.

[29]:
```
# 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

[30]:
```
# Split into training and test
from sklearn.model_selection import train_test_split
Xtr, Xts, ytr, yts = train_test_split(X,y, test_size=0.30)
```

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%.

[31]:
```
from sklearn import datasets, linear_model, preprocessing
reg = linear_model.LogisticRegression(C=1e5)
reg.fit(Xtr, ytr)

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

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

```
[[1.18859414 0.40457683]]
[-5.15102884]
Accuracy on test data = 0.960976
```

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

2.4.13 Question 2a. Gradient descent for logistic regression [20 pts]

In the class we saw that 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

$$L(\mathbf{w}) = -\sum_{i=1}^{N} y_i \log \sigma(\mathbf{w}^T \mathbf{x}_i + b) + (1 - y_i) \log[1 - \sigma(\mathbf{w}^T \mathbf{x}_i + b)].$$

Gradient can be computed as

$$\nabla_{\mathbf{w}} L = \sum_{i=1}^{N} (\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i)\mathbf{x}_i, \quad \nabla_b L = \sum_{i=1}^{N} (\sigma(\mathbf{w}^T \mathbf{x}_i) - y_i).$$

We can update $\mathbf{w}, b$ at every iteration as

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

Note that we could also append the constant term in w and append 1 to every $\mathbf{x}_i$ 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|x,w) = 1/(1+\exp(-z)), z = w[0]+w[1]*x[1] + ... + w[d]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 $L(\mathbf{w})$ 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
```

[32]:
```
# TODO
# Your code for logistic regression via gradient descent goes here
def sigmoid(x):
    return 1/(1 + np.exp(-x))

def compute_cost(X,w,b,y):

    L = 0
    for xi,yi in zip(X,y):
        hyp = w.T@xi + b
        one_pred = sigmoid(hyp)
        one_log = np.log(one_pred)

        zero_pred = 1 - sigmoid(hyp)
        zero_log = np.log(zero_pred)
        l = (yi*one_log) + ((1-yi)*zero_log)
        L += l

    return -L

def logistic_regression_gd(X,y,learning_rate = 0.
↪00001,max_iter=1000,tol=pow(10,-5)):

    w = np.zeros(X[0].shape[0])
    b = 0
    iters = 0
    all_cost = []
    cost = compute_cost(X,w,b,y)
    all_cost.append(cost)

    for epoch in range(max_iter):

        w_now = np.zeros(X[0].shape[0])
        b_now = 0

        for xi,yi in zip(X,y):
            hyp = w.T@xi + b
            pred = sigmoid(hyp)
            err = pred - yi
            w_now += err*xi
            b_now += err

        w -= learning_rate*w_now
        b -= learning_rate*b_now
```

```
        iters += 1
        cost = compute_cost(X,w,b,y)
        all_cost.append(cost)

        if epoch > 1 and np.absolute(all_cost[epoch] - all_cost[epoch-1])/
↪all_cost[epoch-1] <= tol:
            print ('break by tolerance', iters)
            break

    return w, b, all_cost, iters
```

2.4.14 Question 2b: Convergence plots and test accuracy [10 pts]

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 $L(\mathbf{w})$ as a function of the number of iterations, and ideally, what we would like to see is $L(\mathbf{w})$ 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.

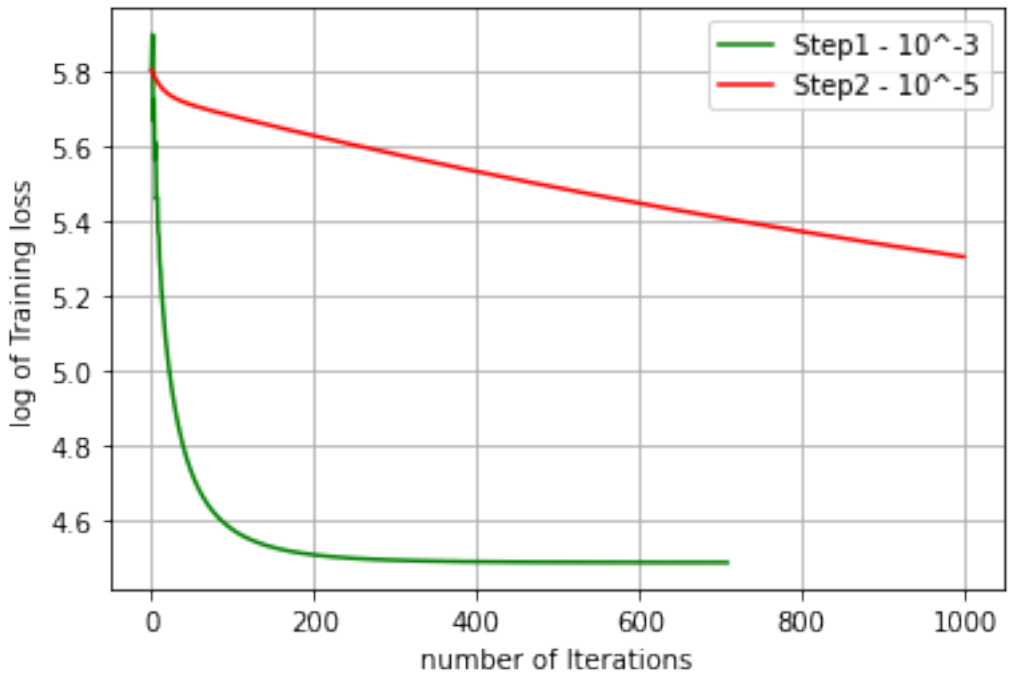
Implement a function `predict(X,w)` that provides you label 1 if $\mathbf{w}^T\mathbf{x} + b > 0$ and 0 otherwise.

[33]:
```
# TODO
# Predict on test samples and measure accuracy
def predict(X,w, b):
    preds = []
    for xi in X:
        grad = w.T@xi+b
        preds.append(grad)

    yhat = [1 if y > 0 else 0 for y in preds]
    return yhat
```

[34]:
```
# TODO
# test gradient descent with step size 0.001
# test gradient descent with step size 0.00001
```

```
(w1, b1,all_cost1,iters1) = logistic_regression_gd(Xtr,ytr,learning_rate = 0.
 ↪001,max_iter = 1000, tol=pow(10,-6))
green, = plt.plot([i for i in range(iters1)], [math.log(i) for i in all_cost1[0:
 ↪iters1]], 'green')
yhat1 = predict(Xts,w1,b1)
acc1 = np.mean(yhat1 == yts)
print("Test accuracy1 = %f" % acc1)

(w2, b2, all_cost2, iters2) = logistic_regression_gd(Xtr,ytr,learning_rate = 0.
 ↪00001,max_iter = 1000, tol=pow(10,-6))
red, = plt.plot([i for i in range(iters2)],[math.log(i) for i in all_cost2[0:
 ↪iters2]],'red')
yhat2 = predict(Xts,w2,b2)
acc2 = np.mean(yhat2 == yts)
print("Test accuracy2 = %f" % acc2)

plt.legend([green, red], ['Step1 - 10^-3', 'Step2 - 10^-5'])
plt.xlabel('number of Iterations')
plt.ylabel('log of Training loss')
plt.grid()
plt.show()
```

```
break by tolerance 709
Test accuracy1 = 0.960976
Test accuracy2 = 0.951220
```



Observations:

1. The implementation with lower step size (red), has not yet converged while the green seems to have converged and reached the defined tolerence.
2. As red has not converged, it will have lower accuracy than green.

2.5 Submission instructions

1. Download this Colab to ipynb, and convert it to PDF. Follow similar steps as here but convert to PDF.

- Download your .ipynb file. You can do it using only Google Colab. `File -> Download -> Download .ipynb`
- Reupload it so Colab can see it. Click on the `Files` icon on the far left to expand the side bar. You can directly drag the downloaded .ipynb file to the area. Or click `Upload to session storage` icon and then select & upload your .ipynb file.
- Conversion using %%shell. `!sudo apt-get update !sudo apt-get install texlive-xetex texlive-fonts-recommended texlive-generic-recommended !jupyter nbconvert --log-level CRITICAL --to pdf name_of_hw.ipynb`
- Your PDF file is ready. Click 3 dots and `Download`.

2. Upload the PDF to Gradescope, select the correct pdf pages for each question. **Important!**
3. Upload the ipynb file to Gradescope

[35]:
```
!sudo apt-get update
!sudo apt-get install texlive-xetex texlive-fonts-recommended␣
↪texlive-generic-recommended
```

```
'sudo' is not recognized as an internal or external command,
operable program or batch file.
'sudo' is not recognized as an internal or external command,
operable program or batch file.
```

[38]:
```
!jupyter nbconvert --log-level CRITICAL --to pdf ./fall2022_hw2.ipynb # make␣
↪sure the ipynb name is correct
```

[34]:
```
```