```
In [ ]:
```

1 Introduction

In this week's exercise session we will learn how to use Machine Learning methods to solve regression problems. In particular, we will focus on linear regression.

First let us import the libraries we will be using during this exercise:

```
In [5]: # Useful starting lines
        %matplotlib inline
        import numpy as np
        np.random.seed(42)
        import matplotlib.pyplot as plt
        import pandas as pd
        %load_ext autoreload
        %autoreload 2
```

```
In [ ]:
```

2 The regression problem

We have seen in class that regression refers to predicting continous values for a given sample. A nice example could be predicting the first salary of a student after graduation given how diligent they were with attending machine learning exercise sessions. We were introduced to the "linear regression method".

Q: How does a regression problem differ from a classification problem?

Q: Why is the linear regression model a linear model?

2.1 Load and inspect the data

This week, using the linear regression method, we will analyze the Boston house prices data set and predict costs based on properties such as per capita crime rate by town, pupil-teacher ratio by town, etc.

We load the data and split it such that 80% and 20% are training and test data, respectively.

In [6]:
```
# get the data set and print a description
import pandas as pd
import numpy as np


data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
target = raw_df.values[1::2, 2]

print(data)
```

```
[[6.3200e-03 1.8000e+01 2.3100e+00 ... 1.5300e+01 3.9690e+02 4.9800e+00]
 [2.7310e-02 0.0000e+00 7.0700e+00 ... 1.7800e+01 3.9690e+02 9.1400e+00]
 [2.7290e-02 0.0000e+00 7.0700e+00 ... 1.7800e+01 3.9283e+02 4.0300e+00]
 ...
 [6.0760e-02 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9690e+02 5.6400e+00]
 [1.0959e-01 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9345e+02 6.4800e+00]
 [4.7410e-02 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9690e+02 7.8800e+00]]
```

In [7]:
```
#les donnees sont dans housing.data.
#X contient toutes les donnes et Y contient la derniere colonne avec le p


#BUT : Trier les données pour :
## enlever les colonnes (=donnes) qui ne nous servent pas pour developper
## meme et la colonne 0

## on ne traite pas les cas des maisons superieurs a 40(000)



X = data
y = target
X = np.delete(X, 3, axis=1) #supprime la 4e colonne qui contient que des

# removing second mode
ind = y<40 # tableau de booleant, contient True si y[0]<40, False sinon.

compteur_false=0

for i in range(len(ind)):
    if ind[i]==False:
        compteur_false=compteur_false+1
print("il y a", compteur_false,"False")

print("longueur de y", len(y))
X = X[ind,:] #on enleve les données dont la 14e valeurs est supérieure a
y = y[ind] #on eleve toutes les prix supérieurs a 40(000)
print("longueur de y", len(y))

print("Shape of the data sample matrix X:", X.shape) #renvoie nb de donné
#la derniere colonne qui est la donnée cible, et la colonne 0, et le nomb
#ou on a elevé les cas des maisons superieures a 40000


print("x_shape=",X.shape)
N = X.shape[0] #renvoie donc le nombre de cas
D = X.shape[1] #renvoie le nombre de données d'entrés : 12
print("The number of data samples is N:", N)
print("The number of features is D:", D)


print("\nShape of the labels vector y:", y.shape)


print("\nFirst data sample in X:", X[0,:]) #renvoie le x1
print("First label in y:", y[0])
```

```
il y a 31 False
longueur de y 506
longueur de y 475
Shape of the data sample matrix X: (475, 12)
x_shape= (475, 12)
The number of data samples is N: 475
The number of features is D: 12

Shape of the labels vector y: (475,)

First data sample in X: [6.320e-03 1.800e+01 2.310e+00 5.380e-01 6.575e+0
0 6.520e+01 4.090e+00
 1.000e+00 2.960e+02 1.530e+01 3.969e+02 4.980e+00]
First label in y: 24.0
```

Our first exercise is to split the data into training and test sets. Let's set aside 80% of the data for training and 20% for testing.

Your steps should be the following ones:

1. Generate a vector of indices from 0 to N-1 (with N being the number of data samples) (Hint: you can use the function np.arange()
2. Shuffle the indices (hint: you can use np.random.shuffle(). Look up how to use this function!)
3. Select 80% of the indices. We have coded this for you but make sure you understand what these lines are doing!

In [8]:
```
split_ratio = 0.8
#80% for the training et 20% pour l'entrainement.
#on a d'abord mélanger tous les indices , puis on a pris les 80 premiers
# pour le test


# Step 1:
indices = np.arange(N)
#creer un tableau commencant a 0 jusqu'a N-1 : 474

np.random.shuffle(indices) #on melange aleatoirement

print(X)
# Step 3:
X_train    = X[indices[0:int(N*split_ratio)] , :]
y_train    = y[indices[0:int(N*split_ratio)]]

print(int(N*split_ratio)) #380 : 80% de 474
#print(indices[0:int(N*split_ratio)] ) # les 380 premieres valeurs du tab

print(X_train) # on a pris X[indices[0]], puis X[indices[1]]...X[indices[
#il s'agit de tableau dans un tableau !


# Split the test data using the remaining indices!


X_test     = X[indices[int(N*split_ratio):],:]#YOUR CODE HERE
y_test     = y[indices[int(N*split_ratio):]] #YOUR CODE HERE
```

```
[[6.3200e-03 1.8000e+01 2.3100e+00 ... 1.5300e+01 3.9690e+02 4.9800e+00]
 [2.7310e-02 0.0000e+00 7.0700e+00 ... 1.7800e+01 3.9690e+02 9.1400e+00]
 [2.7290e-02 0.0000e+00 7.0700e+00 ... 1.7800e+01 3.9283e+02 4.0300e+00]
 ...
 [6.0760e-02 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9690e+02 5.6400e+00]
 [1.0959e-01 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9345e+02 6.4800e+00]
 [4.7410e-02 0.0000e+00 1.1930e+01 ... 2.1000e+01 3.9690e+02 7.8800e+00]]
380
[[2.07162e+01 0.00000e+00 1.81000e+01 ... 2.02000e+01 3.70220e+02
  2.33400e+01]
 [8.24809e+00 0.00000e+00 1.81000e+01 ... 2.02000e+01 3.75870e+02
  1.67400e+01]
 [1.70040e-01 1.25000e+01 7.87000e+00 ... 1.52000e+01 3.86710e+02
  1.71000e+01]
 ...
 [5.73116e+00 0.00000e+00 1.81000e+01 ... 2.02000e+01 3.95280e+02
  7.01000e+00]
 [2.20511e+01 0.00000e+00 1.81000e+01 ... 2.02000e+01 3.91450e+02
  2.21100e+01]
 [4.22239e+00 0.00000e+00 1.81000e+01 ... 2.02000e+01 3.53040e+02
  1.46400e+01]]
```

Check the shapes of `X_train`, `y_train`, `X_test`, `y_test`

In [9]:
```python
print(X_train.shape) #les donnes
print(y_train.shape) #le target
print(X_test.shape) #les 20% de 474 restant
print(y_test.shape)

#shape rend un tuple (nombre de lignes, nombres de colonnes) pour une mat

# The shapes should be (380, 12), (380,), (95, 12), (95,)
```

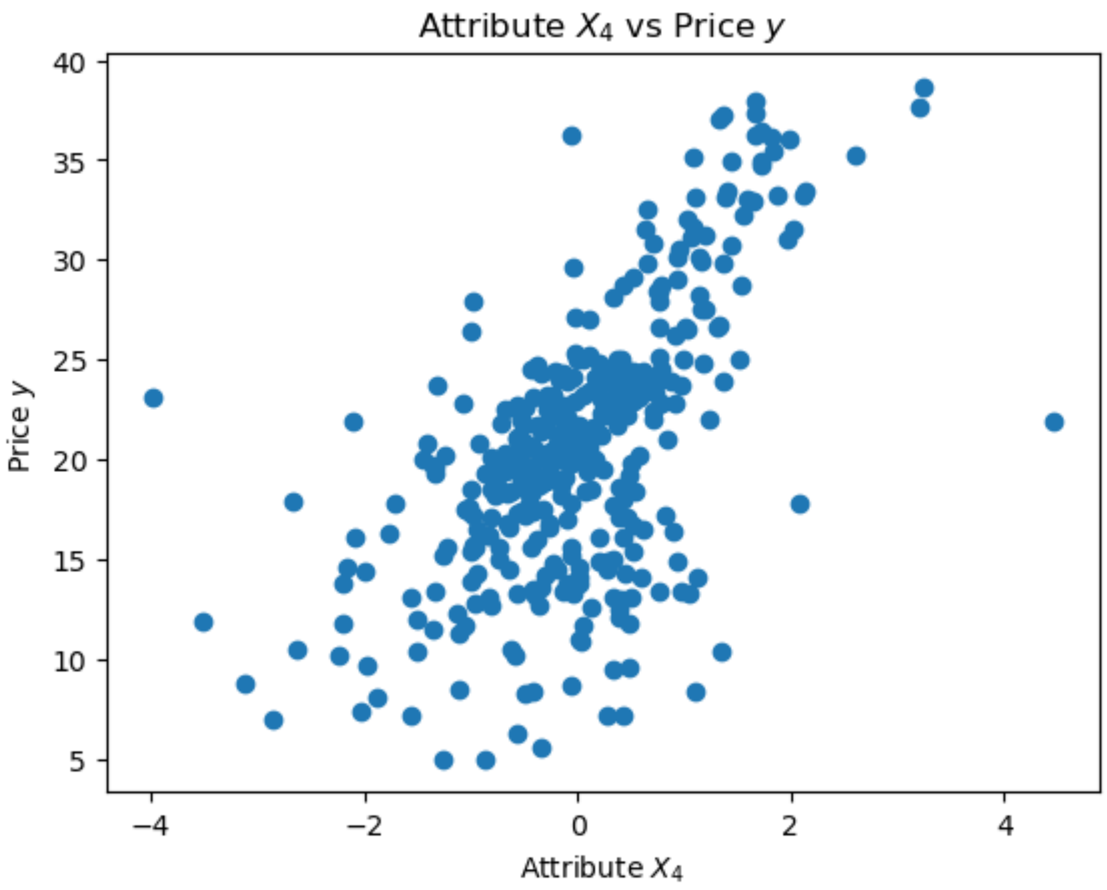
```
(380, 12)
(380,)
(95, 12)
(95,)
```

After, we normalize the data such that each feature has zero mean and unit standard deviation. (Since you have not seen this in the lectures yet, we have provided the code for you. Study this well, we will be doing a lot of normalization during the exercise sessions!)

In [10]:
```python
'''
Make mean 0 and std dev 1 of the data.
'''
def normalize(X):
    mu    = np.mean(X,0,keepdims=True)
    std   = np.std(X,0,keepdims=True)
    X     = (X-mu)/std
    return X, mu, std

#Use train stats for normalizing test set
X_train,mu_train,std_train = normalize(X_train)
X_test = (X_test-mu_train)/std_train
```

In [11]:
```
# Exploratory analysis of the data. Have a look at the distribution of pr

feature = 4
#print(X_train[:,feature]) # 4e colonne (attention on ignore la colonne 0

plt.scatter(X_train[:,feature], y_train) # tracer la 4e colonne en foncti
plt.xlabel(f"Attribute $X_{feature}$")
plt.ylabel("Price $y$")
plt.title(f"Attribute $X_{feature}$ vs Price $y$")
```

Out[11]:
```
Text(0.5, 1.0, 'Attribute $X_4$ vs Price $y$')
```



Q: Using the code above, explore the relation between different features and the house prices. Describe what you see. Can you identify any trends?

We can see that the output y , is linearly correlated to the attribute X4

2.2 Closed-form solution for linear regression

The linear regression method has a closed form, analytical solution, as we have also seen in class.

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

Now let's code the analytical solution in the function `get_w_analytical` and to obtain the weight parameters $\mathbf{w}$. Tip: You may want to use the function np.linalg.pinv().

In [12]:
```
def get_w_analytical(X_train,y_train):
    """
    compute the weight parameters w
    """

    # compute w via the analytical solution
    w = np.linalg.pinv(X_train)@ y_train## Compute the (Moore-Penrose) ps
    print("w vaut",w)
    print("w.shape vaut",w.shape)
    return w

#normalement n nous avons en entrée un vecteur de taille 12 (vecteur colo

#C = A@B renvoie le produit matricielle
#np.linalg.pinv(X_train) renvoie X^TX-1 * X^T

#pas besoin de preciser la transposé , le produit matricielle s'en charge
# w a donc 12 coefficients
```

To assess our method's performance, we'll be using the mean squared error (MSE).

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

where our prediction $\hat{y}_i = \mathbf{x}_i^T \cdot \mathbf{w}$.

Let's code this!

In [13]:
```
def get_loss(w, X_train, y_train,X_test,y_test):

    # le vrai resultat
    loss_train = (1/X_train.shape[0])*np.sum(((X_train@w) - y_train)**2)
    loss_test = (1/X_test.shape[0])*np.sum(((X_test@w) - y_test)**2) # YO
    print("The training loss is {}. The test loss is {}.".format(loss_tra

    return loss_test

# attention: en prenant yi=xi*w, on oublie l'ordonnée a l'origine (= bias
```

In [14]:
```
#Let's test our code!
w_ana = get_w_analytical(X_train,y_train)
get_loss(w_ana, X_train,y_train, X_test,y_test)
```

```
w vaut [-0.96019846  0.7871557  -0.33614233 -1.37813005  1.53507253 -0.52
823449
 -2.28723185  1.97558121 -2.16960627 -1.21073647  0.79146544 -2.49781929]
w.shape vaut (12,)
The training loss is 436.8437185477003. The test loss is 437.329463892094
4.
```

Out[14]:
```
437.3294638920944
```

Q: What is the shape of the analytical weights?

2.3. Adding a bias term

The error of 400 is quite high! Note however that, in contrast to what we have seen in the lectures, we did not use any bias term $\mathrm{w}^{(0)}$. Let's see whether we can reduce the error by including a bias term.

First, let's look more closely at what happens without a bias term. Formally, without a bias term, we are fitting a hyperplane that always passes through the origin. This is because our predictions can be expressed as

$$\hat{y}_i = \mathrm{w}^{(1)} x_i^{(1)} + \mathrm{w}^{(2)} x_i^{(2)} + \ldots + \mathrm{w}^{(12)} x_i^{(12)}$$

or

$$\hat{y}_i = \mathbf{x}_i^T \cdot \mathbf{w}$$

Therefore, when $\mathbf{x}_i = \mathbf{0}$, $\hat{y}_i = 0$, no matter what values $\mathbf{w}$ takes. That's not ideal!

Note: If you are confused about the transpose operation in $\hat{y}_i = \mathbf{x}_i^T \cdot \mathbf{w}$ above, here are the shapes of the matrices that are being multiplied:

- $\mathbf{w}$ is DX1
- $\mathbf{x}_i$ is DX1 (A reminder: The entire data $\mathbf{X}$ is NXD, but when we select a single data sample from it, we express it as a column vector!)
- The result $\hat{y}_i$ is 1x1

Introducing the bias term:

It would be a lot nicer if our predicted hyperplane didn't always have to pass through the origin. In math words:

$$\hat{y}_i = \mathrm{w}^{(0)} + \mathrm{w}^{(1)} x_i^{(1)} + \mathrm{w}^{(2)} x_i^{(2)} + \ldots + \mathrm{w}^{(12)} x_i^{(12)}$$

Here, the $\mathrm{w}^{(0)}$ is the y-intercept. When $\mathbf{x}_i = \mathbf{0}$, $y_i = \mathrm{w}^{(0)}$. Neat!

To handle this, we can add a column of 1s as a feature in our data $\mathbf{X}$. This way, we could just say that the last feature $x_i^{(0)} = 1$ and

$$\hat{y}_i = \mathrm{w}^{(0)} \cdot 1 + \mathrm{w}^{(1)} x_i^{(1)} + \mathrm{w}^{(2)} x_i^{(2)} + \ldots + \mathrm{w}^{(12)} x_i^{(12)}$$

$$\hat{y}_i = \mathrm{w}^{(0)} x_i^{(0)} + \mathrm{w}^{(1)} x_i^{(1)} + \mathrm{w}^{(2)} x_i^{(2)} + \ldots + \mathrm{w}^{(12)} x_i^{(12)}$$

$$\hat{y}_i = \mathbf{x}_i^T \cdot \mathbf{w}$$

And we can keep using the same analytical solution formula as above! So by adding a column of 1s as the last feature of $\mathbf{X}$, and running the analytical solution, we will find a $\mathbf{w}$ with 13 features instead of 12. The last feature of the weights $\mathbf{w}^{(0)}$ will be the bias term. This way, we wouldn't have to change any of the functions we wrote above.

So let's get to it! Fill in the function below to append a bias term to the data matrices $\mathbf{X}$. Your steps should be the following:

1. Create a numpy array that is a column of 1s. It's shape should be NX1.
2. Concatenate the ones column with the data matrix. Hint: use np.concatenate. Be careful what axis you specify!

In [15]:
```
def append_bias_term(X_train):

    ones_column = np.ones([X_train.shape[0],1])# YOUR CODE HERE
    X_train_bias = np.concatenate((X_train,ones_column),axis=1)# YOUR COD
    return X_train_bias
```

In [16]:
```
X_train_bias = append_bias_term(X_train)
X_test_bias = append_bias_term(X_test)

w_ana = get_w_analytical(X_train_bias,y_train)

get_loss(w_ana, X_train_bias,y_train, X_test_bias,y_test)
```

```
w vaut [-0.96019846  0.7871557  -0.33614233 -1.37813005  1.53507253 -0.52
823449
 -2.28723185  1.97558121 -2.16960627 -1.21073647  0.79146544 -2.49781929
 20.65052632]
w.shape vaut (13,)
The training loss is 10.399481428585586. The test loss is 11.792470347125
78.
```

Out[16]:
```
11.79247034712578
```

Now your loss should be around 10. That's much better, no?

2.4. Solution using gradient descent

The linear regression model has an analytical solution, but we can also get the weight parameters $\mathbf{w}$ numerically, e.g., via gradient descent. We will be using this approach to complete the function `get_w_numerical` below.

First, let us code the gradient of the MSE loss, as we saw in class:

$$\nabla R = \frac{2}{N}\sum_{i}^{N}(\mathbf{x}_i^T\mathbf{w} - y_i)\cdot\mathbf{x}_i$$

In [17]:
```
def find_gradient(X, y, w):
    """computes the gradient of the empirical risk, R"""

    N = X.shape[0]


    # YOUR CODE HERE

    grad = (2/N)*(((X@w) - y)@X) # YOUR CODE HERE
    return grad
```

In [18]:
```
#Let's create a random w to test the function above.
w = np.random.normal(0, 1e-1, X_train_bias.shape[1])
grad = find_gradient(X_train_bias, y_train, w)
print(grad.shape)
```

```
(13,)
```

And now, we can write the function that finds $\mathbf{w}$ using gradient descent. Recall that gradient descent works via the update

$$w_k \leftarrow w_{k-1} - \eta\nabla R$$

where η is the learning rate and k is the iteration number.

Fill in the function below to update $\mathbf{w}$.

In [19]:
```
def get_w_numerical(X_train,y_train,X_test,y_test,epochs,lr):
    """compute the weight parameters w"""

    # initialize the weights
    w = np.random.normal(0, 1e-1, X_train.shape[1])

    # iterate a given number of epochs over the training data
    for epoch in range(epochs):

        w = w - (lr*find_gradient(X_train,y_train,w))# YOUR CODE HERE

        # Test every 500 epochs to see whether the training loss and test
        if epoch % 500 == 0:
            print(f"\nEpoch {1000+epoch}/{epochs}")
            get_loss(w, X_train,y_train, X_test,y_test)

    return w
```

In [20]:
```
# compute w and calculate its performance
w_num = get_w_numerical(X_train_bias,y_train,X_test_bias,y_test,15000,1e-
Epoch 1000/15000
The training loss is 466.80255417095094. The test loss is 525.93485103716
19.

Epoch 1500/15000
The training loss is 70.00185952395204. The test loss is 73.6981894036700
1.

Epoch 2000/15000
The training loss is 19.404327246296788. The test loss is 21.860606540981
244.

Epoch 2500/15000
The training loss is 12.330178826265604. The test loss is 14.515899448314
585.

Epoch 3000/15000
The training loss is 11.208547509516823. The test loss is 13.269249537410
559.

Epoch 3500/15000
The training loss is 10.938330929164872. The test loss is 12.903862822878
033.

Epoch 4000/15000
The training loss is 10.814610882591744. The test loss is 12.700093445829
575.

Epoch 4500/15000
The training loss is 10.7323807769805. The test loss is 12.55071180193240
6.

Epoch 5000/15000
The training loss is 10.671137576908023. The test loss is 12.433542208256
432.

Epoch 5500/15000
The training loss is 10.623869967267419. The test loss is 12.339692731512
985.

Epoch 6000/15000
The training loss is 10.58670154182522. The test loss is 12.2635236755631
87.

Epoch 6500/15000
The training loss is 10.557043764032477. The test loss is 12.200967244898
328.

Epoch 7000/15000
The training loss is 10.533068218282672. The test loss is 12.149016162335
515.

Epoch 7500/15000
The training loss is 10.513454780670322. The test loss is 12.105429047246
306.

Epoch 8000/15000
The training loss is 10.49723680986379. The test loss is 12.0685214349993
61.

Epoch 8500/15000
The training loss is 10.483697707229792. The test loss is 12.037014238788
762.
```

```
Epoch 9000/15000
The training loss is 10.47229973670482. The test loss is 12.0099245994463
15.

Epoch 9500/15000
The training loss is 10.46263433760145. The test loss is 11.9864876518644
93.

Epoch 10000/15000
The training loss is 10.45438711352743. The test loss is 11.9661004648138
82.

Epoch 10500/15000
The training loss is 10.4473129769489. The test loss is 11.94828173115325
3.

Epoch 11000/15000
The training loss is 10.441218382435661. The test loss is 11.932642605016
353.

Epoch 11500/15000
The training loss is 10.435948539574902. The test loss is 11.918865430574
717.

Epoch 12000/15000
The training loss is 10.431378142555975. The test loss is 11.906688074864
228.

Epoch 12500/15000
The training loss is 10.427404595383278. The test loss is 11.895892260685
692.

Epoch 13000/15000
The training loss is 10.423943016863335. The test loss is 11.886294774187
174.

Epoch 13500/15000
The training loss is 10.420922521583526. The test loss is 11.877740755507
169.

Epoch 14000/15000
The training loss is 10.418283421115326. The test loss is 11.870098513339
448.

Epoch 14500/15000
The training loss is 10.415975093323878. The test loss is 11.863255466309
772.

Epoch 15000/15000
The training loss is 10.413954340444773. The test loss is 11.857114927210
295.

Epoch 15500/15000
The training loss is 10.412184107818481. The test loss is 11.851593525417
238.
```

If everything went well, then your loss should be going down with each epoch!

Q: How do these results compare to those of the analytical solution?

Q: In which cases may it be preferable to use the numerical approach over the analytical solution?

it's equal to the analytical solution with the bias if we have a lot of data (huge matrix) numerical is better

2.5 Using sklearn

We can also use the sklearn implementation of the linear regression model. sklearn is a library that contains implementations of many popular machine learning models, including the linear regression model!

Please look up the documentation to

1. instantiate the LinearRegression model
2. fit the model to our training data
3. evaluate the model on the test data
4. and compare the results with our previous outcomes

Especially check out the example code they provide!

In [21]:
```
from sklearn.linear_model import LinearRegression
from sklearn import metrics

# Create linear regression object   # .fit = Train the model using the tr
reg = LinearRegression().fit(X_train,y_train)
#Return the coefficient of determination of the prediction
reg.score(X_train,y_train)
print("Coefficients: \n", reg.coef_)
reg.intercept_
# Make predictions using the testing set
y_hat = reg.predict(X_test)
# YOUR CODE HERE

# The mean squared error
print('MSE of sklearn linear regression model on test data: ' , metrics.m
```

```
Coefficients: 
 [-0.96019846  0.7871557  -0.33614233 -1.37813005  1.53507253 -0.52823449
 -2.28723185  1.97558121 -2.16960627 -1.21073647  0.79146544 -2.49781929]
MSE of sklearn linear regression model on test data:  11.792470347125802
```

In [ ]:
```
```