Linear Regression one variable (manual implementation)

Import necessary packages

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

Set interactive backend

```
%matplotlib inline
```

Load data set

```
'''
The load_boston() command was removed from the scikit-learn library
after version 1.2 (I tried install older version but it gives errors)
So, i decided to import dataset boston directly from server. ( as has
been suggested by scikit-learn devs)

from sklearn.datasets import load_boston
boston = load_boston()
df = pd.DataFrame (boston.data, columns=boston.feature_names)
y = boston.target

'''

# Importing directly
```

```
data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
data1 = raw_df.values[::2, :]
data2 = raw_df.values[1::2, :2]
data = np.hstack((data1, data2))
target = raw_df.values[1::2, 2]

# Creating Dataframe
df = pd.DataFrame(data, columns=["CRIM", "ZN", "INDUS", "CHAS",
                  "NOX", "RM", "AGE", "DIS", "RAD", "TAX", "PTRATIO",
"B", "LSTAT"])
y = target

df
```

```
<>:14: SyntaxWarning: invalid escape sequence '\s'
<>:14: SyntaxWarning: invalid escape sequence '\s'
C:\Users\Smeek\AppData\Local\Temp\ipykernel_4056\2184707510.py:14:
SyntaxWarning: invalid escape sequence '\s'
  raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
```

```
        CRIM    ZN  INDUS  CHAS    NOX     RM   AGE     DIS  RAD
TAX  \
0    0.00632  18.0   2.31   0.0  0.538  6.575  65.2  4.0900  1.0
296.0
1    0.02731   0.0   7.07   0.0  0.469  6.421  78.9  4.9671  2.0
242.0
2    0.02729   0.0   7.07   0.0  0.469  7.185  61.1  4.9671  2.0
242.0
3    0.03237   0.0   2.18   0.0  0.458  6.998  45.8  6.0622  3.0
222.0
4    0.06905   0.0   2.18   0.0  0.458  7.147  54.2  6.0622  3.0
222.0
..       ...   ...    ...   ...    ...    ...   ...     ...  ...    ..
.
501  0.06263   0.0  11.93   0.0  0.573  6.593  69.1  2.4786  1.0
273.0
502  0.04527   0.0  11.93   0.0  0.573  6.120  76.7  2.2875  1.0
273.0
503  0.06076   0.0  11.93   0.0  0.573  6.976  91.0  2.1675  1.0
273.0
504  0.10959   0.0  11.93   0.0  0.573  6.794  89.3  2.3889  1.0
273.0
505  0.04741   0.0  11.93   0.0  0.573  6.030  80.8  2.5050  1.0
273.0

    PTRATIO       B  LSTAT
0      15.3  396.90   4.98
1      17.8  396.90   9.14
2      17.8  392.83   4.03
```

```
3        18.7  394.63   2.94
4        18.7  396.90   5.33
..        ...     ...    ...
501      21.0  391.99   9.67
502      21.0  396.90   9.08
503      21.0  396.90   5.64
504      21.0  393.45   6.48
505      21.0  396.90   7.88

[506 rows x 13 columns]
```

Warning does not affect the final result

Select one feature

```
df = df[['RM']] # Note: returns df comparing to  df['RM']
df['target'] = y

C:\Users\Smeek\AppData\Local\Temp\ipykernel_4056\1082073822.py:2:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation:
https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#
returning-a-view-versus-a-copy
  df['target'] = y
```

Review the data

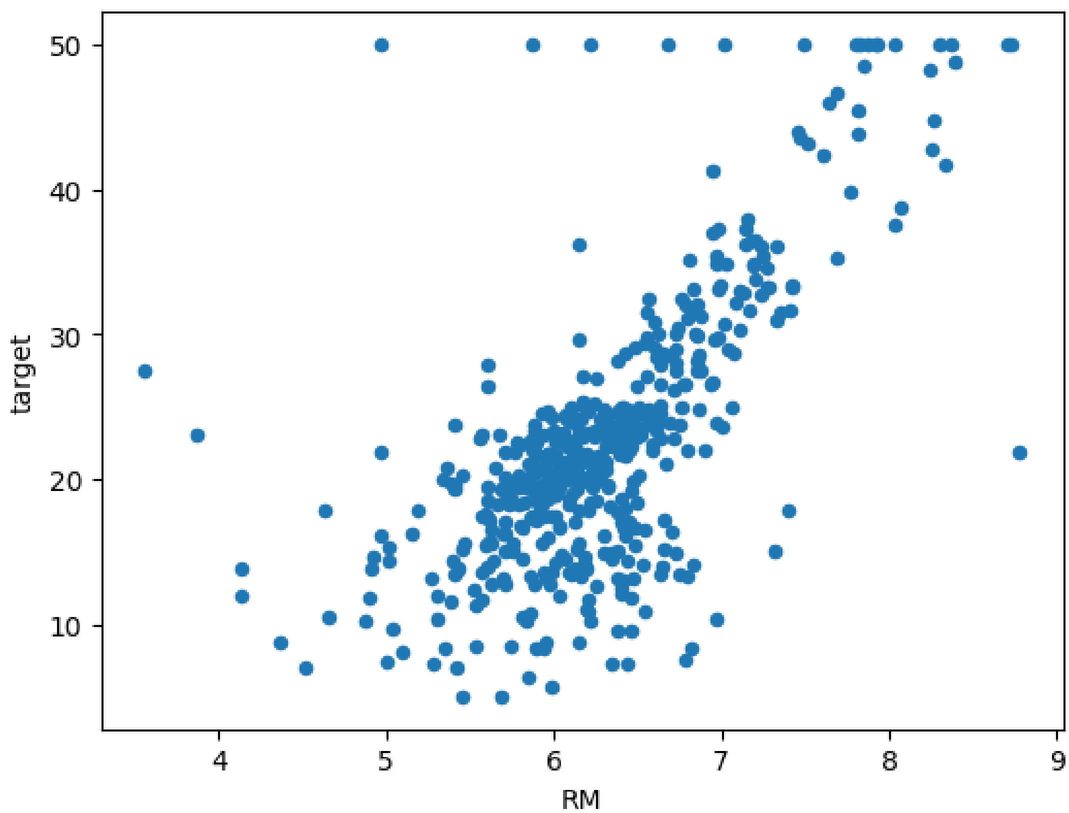
```
print (df.head(10))
df.describe ()

      RM  target
0  6.575    24.0
1  6.421    21.6
2  7.185    34.7
3  6.998    33.4
4  7.147    36.2
5  6.430    28.7
6  6.012    22.9
```

```
7  6.172    27.1
8  5.631    16.5
9  6.004    18.9
```

	RM	target
count	506.000000	506.000000
mean	6.284634	22.532806
std	0.702617	9.197104
min	3.561000	5.000000
25%	5.885500	17.025000
50%	6.208500	21.200000
75%	6.623500	25.000000
max	8.780000	50.000000

```
df.plot.scatter('RM', 'target')
```

```
<Axes: xlabel='RM', ylabel='target'>
```



Custom Linear Regression Classifier

Load all data

```
'''
X, y = load_boston(return_X_y=True)
Зновуж таки завантажимо інформацію вручну
'''
# Завантаження даних вручну з прямого посилання
data_url = "http://lib.stat.cmu.edu/datasets/boston"
raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
data1 = raw_df.values[::2, :]
data2 = raw_df.values[1::2, :2]
data = np.hstack((data1, data2))
target = raw_df.values[1::2, 2]

# Створення DataFrame для даних та цільової змінної
X = pd.DataFrame(data, columns=["CRIM", "ZN", "INDUS", "CHAS", "NOX",
                 "RM", "AGE", "DIS", "RAD", "TAX", "PTRATIO", "B",
"LSTAT"])
y = target


# YOUR_CODE. select the values of feature 5 only (corresponding to
'RM') and assign to X
# START_CODE
X = df['RM'].values
# END_CODE

X = X.reshape(-1, 1)  # make it 2d as for case of mutivariable

# YOUR_CODE. Apply train_test_split to X and Y to get X_train, X_test,
y_train, y_test
# START_CODE
X_train, X_test, y_train, y_test = train_test_split(X, y)
# END_CODE
```

```
<>:7: SyntaxWarning: invalid escape sequence '\s'
<>:7: SyntaxWarning: invalid escape sequence '\s'
C:\Users\Smeek\AppData\Local\Temp\ipykernel_4056\4023815593.py:7:
SyntaxWarning: invalid escape sequence '\s'
  raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
```

Check loaded data

```
# DON'T_CHANGE_THIS_CODE. It is used to let you check the result is correct
print ('X_train.shape= ',X_train.shape)
print ('y_train.shape= ',y_train.shape)
X_train[:10]
```

```
X_train.shape=  (379, 1)
y_train.shape=  (379,)
```

```
array([[6.301],
       [7.831],
       [6.816],
       [6.861],
       [5.713],
       [5.682],
       [6.315],
       [5.   ],
       [6.127],
       [5.813]])
```

Expected output:

`X_train.shape=  (379, 1)` `y_train.shape=  (379,)`

Develop expresion of h

```
class Linear_Regression_1():
    def __init__(self):
        pass

    def h(self, b, w, X):
        '''
        :param b -  float or ndarry of shape [m,1], m - number of samples
        :param w - ndarray of shape [1,n],  n - number of features
        :param X - ndarray of shape [m,n], m - number of samples, n - number of features
        '''
        assert (X.shape[1]== w.shape[1])

        # YOUR_CODE. Assign expression for h to h_res
        # START_CODE
        h_res= b + np.dot(X, w.T)
        # END_CODE

        return h_res
```

Check h

```
# DON'T_CHANGE_THIS_CODE. It is used to let you check the result is correct
np.random.seed(2018)
b_check= np.random.randn()
w_check= np.random.randn(1,1)
X_check= np.random.randn(10,1)
print('b= {}, \nw= {}, \nX= \n{}'.format(b_check, w_check, X_check))
lin_reg_1 = Linear_Regression_1()
lin_reg_1.h(b_check, w_check, X_check)
```

```
b= -0.276767596147759,
w= [[0.581851]],
X=
[[ 2.14839926]
 [-1.279487  ]
 [ 0.50227689]
 [ 0.8560293 ]
 [-0.14279008]
 [ 0.11007867]
 [-0.68806479]
 [ 0.43356408]
 [ 0.510221  ]
 [-0.16513097]]
```

```
array([[ 0.97328067],
       [-1.02123839],
       [ 0.01548272],
       [ 0.22131391],
       [-0.35985014],
       [-0.21271821],
       [-0.67711878],
       [-0.0244979 ],
       [ 0.02010501],
       [-0.37284922]])
```

Expected output:

``` ([[ 0.97328067], [-1.02123839], [ 0.01548272], [ 0.22131391], [-0.35985014], [-0.21271821], [-0.67711878], [-0.0244979 ], [ 0.02010501], [-0.37284922]])

## Develop expresion of Cost Function
```

```
class Linear_Regression_2():
    '''linear regression using gradient descent
    '''
    def __init__(self):
        pass


    def J (self, h, y):
        '''
        :param h - ndarray of shape (m,1)
        :param y - ndarray of shape (m,1)
        :return expression for cost function
        '''
        if h.shape !=y.shape:
            print('h.shape = {} does not match y.shape = {}.Expected
{}'.format (h.shape, y.shape, (self.m,1)))
            raise Exception('Check assertion in J')

        # YOUR_CODE. Assign expression for J to J_res
        # START_CODE
        m = h.shape[0]
        err = h - y
        J_res = (1 / (2 * m)) * np.dot(err.T, err)
        # END_CODE
        return J_res
```

Check J

```
# DON'T_CHANGE_THIS_CODE. It is used to let you check the result is
correct
np.random.seed(2019)
m = 10
y_check= np.random.randn(m,1)
h_check= np.random.randn(m,1)
print('y= {}, \nh= {}'.format(y_check, h_check))
lin_reg_2 = Linear_Regression_2()
lin_reg_2.m = m
lin_reg_2.J(h_check, y_check)
```

```
y= [[-0.21767896]
 [ 0.82145535]
 [ 1.48127781]
 [ 1.33186404]
 [-0.36186537]
 [ 0.68560883]
```

```
 [ 0.57376143]
 [ 0.28772767]
 [-0.23563426]
 [ 0.95349024]],
h= [[-1.6896253 ]
 [-0.34494271]
 [ 0.0169049 ]
 [-0.51498352]
 [ 0.24450929]
 [-0.18931261]
 [ 2.67217242]
 [ 0.46480249]
 [ 0.84593044]
 [-0.50354158]]

array([[0.89714652]])
```

Expected output:

```
0.897146515186598
```

Develop expresion of Cost Function derivative

```
class Linear_Regression_3():
    def __init__(self, max_iter = 1e5, alpha = 1,eps = 1e-10, verbose=
0):
        pass

    def h(self, b, w, X):
        '''
        :param b -  float or ndarry of shape [m,1], m - number of
samples
        :param w - ndarray of shape [1,m],  n - number of features
        :param X - ndarray of shape [m,n], m - number of samples, n -
number of features
        '''
        assert (X.shape[1]== w.shape[1])

        # YOUR_CODE. Insert the expression of h developed in
Linear_Regression_1
        # START_CODE
        h_res= b + np.dot(X, w.T)
        # END_CODE

        return h_res
```

```
    def J_derivative(self, params, X, y):
        '''
        :param params - tuple (b,w), where w is the 2d ndarry of shape
(1,n), n- number of features
        :param X- ndarray of shape (m, n)
        :param y - ndarray of shape (m,1)
        :return tuple of derivatrives of cost function by b and w
        '''
        b,w = params
        assert (w.shape == (1,self.n))
        h_val = self.h(b,w,X)
        if  h_val.shape != (self.m, 1):
            print('h.shape = {}, but expected {}'.format (h_val.shape,
(self.m, 1)))
            raise Exception('Check assertion in J_derivative')

        # YOUR_CODE. Assign expressions for derivates of J by b and by
w  to dJ_b and dJ_w corrrespondingly
        # START_CODE
        dJ_b = np.sum(h_val - y) / self.m
        dJ_w = np.dot((h_val - y).T, X) / self.m
        # END_CODE

        return (dJ_b, dJ_w)
```

Check cost function derivatives

```
# DON'T_CHANGE_THIS_CODE. It is used to let you check the result is
correct
np.random.seed(2020)
m = 10
n = 1
X_check= np.random.randn(m,n)
y_check= np.random.randn(m,1)
b_check= np.random.randn()
w_check= np.random.randn(1,n)
params = b_check,w_check
print('X= {}, \ny= {}, \nb= {} \nw= {}'.format(X_check, y_check,
b_check, w_check))

lin_reg_3 = Linear_Regression_3()
lin_reg_3.m = m
lin_reg_3.n = n
lin_reg_3.J_derivative(params, X_check, y_check)
```

```
X= [[-1.76884571]
 [ 0.07555227]
 [-1.1306297 ]
 [-0.65143017]
 [-0.89311563]
 [-1.27410098]
 [-0.06115443]
 [ 0.06451384]
 [ 0.41011295]
 [-0.57288249]],
y= [[-0.80133362]
 [ 1.31203519]
 [ 1.27469887]
 [-1.2143576 ]
 [ 0.31371941]
 [-1.44482142]
 [-0.3689613 ]
 [-0.76922658]
 [ 0.3926161 ]
 [ 0.05729383]],
b= 2.0899788404287745
w= [[0.04197131]]

(2.190460881995871, array([[-1.43284262]]))
```

Expected output:

```
(2.1904608819958713, -1.4328426209410612)
```

Develop gradient descent

```
class Linear_Regression_4():
    '''
    linear regression using gradient descent
    '''
    def __init__(self, max_iter = 1e5, alpha = 0.01,eps = 1e-10,
verbose= 0):
        '''
        :param verbose: set 1 to display more details of J val changes
        '''
        self.max_iter = max_iter
        self.alpha = alpha
        self.eps = eps
        self.verbose = verbose

    def h(self, b, w, X):
```

```
        '''
        :param b -  float or ndarry of shape [m,1], m - number of
samples
        :param w - ndarray of shape [1,n],  n - number of features
        :param X - ndarray of shape [m,n], m - number of samples, n -
number of features
        '''
        assert (X.shape[1]== w.shape[1])

        # YOUR_CODE. Insert the expression of h developed in
Linear_Regression_1
        # START_CODE
        h_res= b + np.dot(X, w.T)
        # END_CODE

        if h_res.shape != (X.shape[0],1):
            print('h.shape = {} but expected {}'.format (h_res.shape,
(self.m,1)))
            raise Exception('Check assertion in h')
        return h_res

    def J (self, h, y):
        '''
        :param h - ndarray of shape (m,1)
        :param y - ndarray of shape (m,1)
        :return expression for cost function
        '''
        if h.shape !=y.shape:
            print('h.shape = {} does not match y.shape = {}.Expected
{}'.format (h.shape, y.shape, (self.m,1)))
            raise Exception('Check assertion in J')
        # YOUR_CODE. Insert the expression of J developed in
Linear_Regression_2
        # START_CODE
        m = h.shape[0]
        err = h - y
        J_res = (1 / (2 * m)) * np.dot(err.T, err)
        # END_CODE

        return J_res

    def J_derivative(self, params, X, y):
        '''
        :param params - tuple (b,w), where w is the 2d ndarry of shape
(1,n), n- number of features
        :param X- ndarray of shape (m, n)
        :param y - ndarray of shape (m,1)
        :return tuple of derivatrives of cost function by b and w
        '''
```

```
        b,w = params
        assert (w.shape == (1,self.n))
        h_val = self.h(b,w,X)
        if  h_val.shape != (self.m, 1):
            print('h.shape = {}, but expected {}'.format (h_val.shape,
(self.m, 1)))
            raise Exception('Check assertion in J_derivative')

        # YOUR_CODE. Insert the expressions for derivates of J by b
and by w to dJ_b and dJ_w developed in Linear_Regression_3
        # START_CODE
        dJ_b = np.sum(h_val - y) / self.m
        dJ_w = np.dot((h_val - y).T, X) / self.m
        # END_CODE

        return (dJ_b, dJ_w)

    def fit(self, X, y):
        '''
        :param X - ndarray training set of shape [m,n], m - number of
samples, n - number of features
        :param y - ndarray - 1d array
        :return: True in case of successful fit
        '''
        if self.verbose:
            print ('Running gradient descent with alpha = {}, eps= {},
max_iter= {}'.format(
                self.alpha, self.eps, self.max_iter))
        self.m,self.n= X.shape # number of samples, number of features

        y = y.reshape(self.m,1) # make it 2 d to make sure it
corresponds to h_val
        b = 0 # init intercept with 0
        w= np.zeros(self.n).reshape(1,-1) # make sure it's shape is
[1,n]
        params = (b,w)

        self.J_hist=[-1] # used for keeping J values. Init with -1 to
avoid 0 at first iter
        continue_iter = True # flag to continue next iter (grad desc
step)
        iter_number =0 # used for limit by max_iter

        while continue_iter:
            # Do step of gradient descent
            # YOUR_CODE. Develop one step of gradien descent
            # START_CODE
            dJ_b, dJ_w = self.J_derivative(params, X, y)
            b -= self.alpha * dJ_b
            w -= self.alpha * dJ_w
```

```
            params = (b, w)
            # END_CODE

            # keep history of J values
            self.J_hist.append(self.J(self.h(b, w, X), y))
            if self.verbose:
                print ('b = {}, w= {}, J= {}'.format(b,w,self.J_hist[-
1]))
            # check criteria of exit the loop (finish grad desc)
            if self.max_iter and iter_number> self.max_iter: # if
max_iter is provided and limit succeeded
                continue_iter = False
            elif np.abs(self.J_hist[iter_number-1] -
self.J_hist[iter_number])< self.eps: # if accuracy is succeeded
                continue_iter = False
            iter_number += 1

        # store the final params to further using
        self.intercept_, self.coef_= params
        return True
```

Check gradient descent

```
# DON'T_CHANGE_THIS_CODE. It is used to let you check the result is
correct
np.random.seed(2021)
m = 10
n = 1
X_check= np.random.randn(m,n)
y_check= np.random.randn(m,1)
print('X= {}, \ny= {}'.format(X_check, y_check))
lin_reg_4 = Linear_Regression_4(alpha = 1, max_iter = 5, verbose=1)
lin_reg_4.fit(X_check, y_check)
```

```
X= [[ 1.48860905]
 [ 0.67601087]
 [-0.41845137]
 [-0.80652081]
 [ 0.55587583]
 [-0.70550429]
 [ 1.13085826]
 [ 0.64500184]
 [ 0.10641374]
 [ 0.42215483]],
```

```
y= [[ 0.12420684]
 [-0.83795346]
 [ 0.4090157 ]
 [ 0.10275122]
 [-1.90772239]
 [ 1.1002243 ]
 [-1.40232506]
 [-0.22508127]
 [-1.33620597]
 [ 0.30372151]]
Running gradient descent with alpha = 1, eps= 1e-10, max_iter= 5
b = -0.3669368558728844, w= [[-0.4217246]], J= [[0.33976525]]
b = -0.23643637277401236, w= [[-0.46886908]], J= [[0.3278115]]
b = -0.22184776004990137, w= [[-0.52721539]], J= [[0.32509097]]
b = -0.20379279582278398, w= [[-0.55396166]], J= [[0.32428458]]
b = -0.19551630227029396, w= [[-0.5697399]], J= [[0.32403801]]
b = -0.19063380881762437, w= [[-0.57831305]], J= [[0.32396239]]
b = -0.18798089094052142, w= [[-0.58309057]], J= [[0.32393919]]

True
```

Expected output:

```
Running gradient descent with alpha = 1, eps= 1e-10, max_iter= 5
b = -0.36693685587288444, w= [[-0.4217246]], J= 0.33976525493056825
b = -0.23643637277401236, w= [[-0.46886908]], J= 0.3278115023016167
b = -0.22184776004990137, w= [[-0.52721539]], J= 0.3250909705515032
b = -0.20379279582278398, w= [[-0.55396166]], J= 0.32428457786538833
b = -0.19551630227029396, w= [[-0.5697399]], J= 0.32403801171263197
b = -0.19063380881762437, w= [[-0.57831305]], J= 0.3239623872203208
b = -0.18798089094052142, w= [[-0.58309057]], J= 0.3239391853771439
```

Alltogether

Please copy the code of functions you developed above to the class corresponding functions.

Please review addtional already implemented functions: draw_cost_changes(), predict() and score()

```
from sklearn.metrics import r2_score
class Linear_Regression():
    '''
    linear regression using gradient descent
    '''
    def __init__(self, max_iter = 1e5, alpha = 0.01, eps = 1e-10,
```

```
verbose= 0):
        '''
        :param verbose: set 1 to display more details of J val changes
        '''
        self.max_iter = max_iter
        self.alpha = alpha
        self.eps = eps
        self.verbose = verbose

    def h(self, b, w, X):
        '''
        :param b -  float or ndarry of shape [m,1], m - number of
samples
        :param w - ndarray of shape [1,m],  n - number of features
        :param X - ndarray of shape [m,n], m - number of samples, n -
number of features
        '''
        assert (X.shape[1]== w.shape[1])

        # YOUR_CODE. Insert the expression of h developed in
Linear_Regression_1
        # START_CODE
        h_res= b + np.dot(X, w.T)
        # END_CODE

        if h_res.shape != (X.shape[0],1):
            print('h.shape = {} but expected {}'.format (h_res.shape,
(self.m,1)))
            raise Exception('Check assertion in h')
        return h_res

    def J (self, h, y):
        '''
        :param h - ndarray of shape (m,1)
        :param y - ndarray of shape (m,1)
        :return expression for cost function
        '''
        if h.shape !=y.shape:
            print('h.shape = {} does not match y.shape = {}.Expected
{}'.format (h.shape, y.shape, (self.m,1)))
            raise Exception('Check assertion in J')
        # YOUR_CODE. Insert the expression of J developed in
Linear_Regression_2
        # START_CODE
        m = h.shape[0]
        err = h - y
        J_res = (1 / (2 * m)) * np.dot(err.T, err)
        # END_CODE

        return J_res
```

```
    def J_derivative(self, params, X, y):
        '''
        :param params - tuple (b,w), where w is the 2d ndarry of shape
(1,n), n- number of features
        :param X- ndarray of shape (m, n)
        :param y - ndarray of shape (m,1)
        :return tuple of derivatrives of cost function by b and w
        '''

        b,w = params
        assert (w.shape == (1,self.n))
        h_val = self.h(b,w,X)
        if  h_val.shape != (self.m, 1):
            print('h.shape = {}, but expected {}'.format (h_val.shape,
(self.m, 1)))
            raise Exception('Check assertion in J_derivative')

        # YOUR_CODE. Insert the expressions for derivates of J by b
and by w to dJ_b and dJ_w developed in Linear_Regression_3
        # START_CODE
        dJ_b = np.sum(h_val - y) / self.m
        dJ_w = np.dot((h_val - y).T, X) / self.m
        # END_CODE

        return (dJ_b, dJ_w)

    def fit(self, X, y):
        '''
        :param X - ndarray training set of shape [m,n], m - number of
samples, n - number of features
        :param y - ndarray - 1d array
        :return: True in case of successful fit
        '''
        if self.verbose:
            print ('Running gradient descent with alpha = {}, eps= {},
max_iter= {}'.format(
                self.alpha, self.eps, self.max_iter))
        self.m,self.n= X.shape # number of samples, number of features

        y = y.reshape(self.m,1) # make it 2 d to make sure it
corresponds to h_val
        b = 0 # init intercept with 0
        w= np.zeros(self.n).reshape(1,-1) # make sure it's shape is
[1,n]
        params = (b,w)

        self.J_hist=[-1] # used for keeping J values. Init with -1 to
avoid 0 at first iter
```

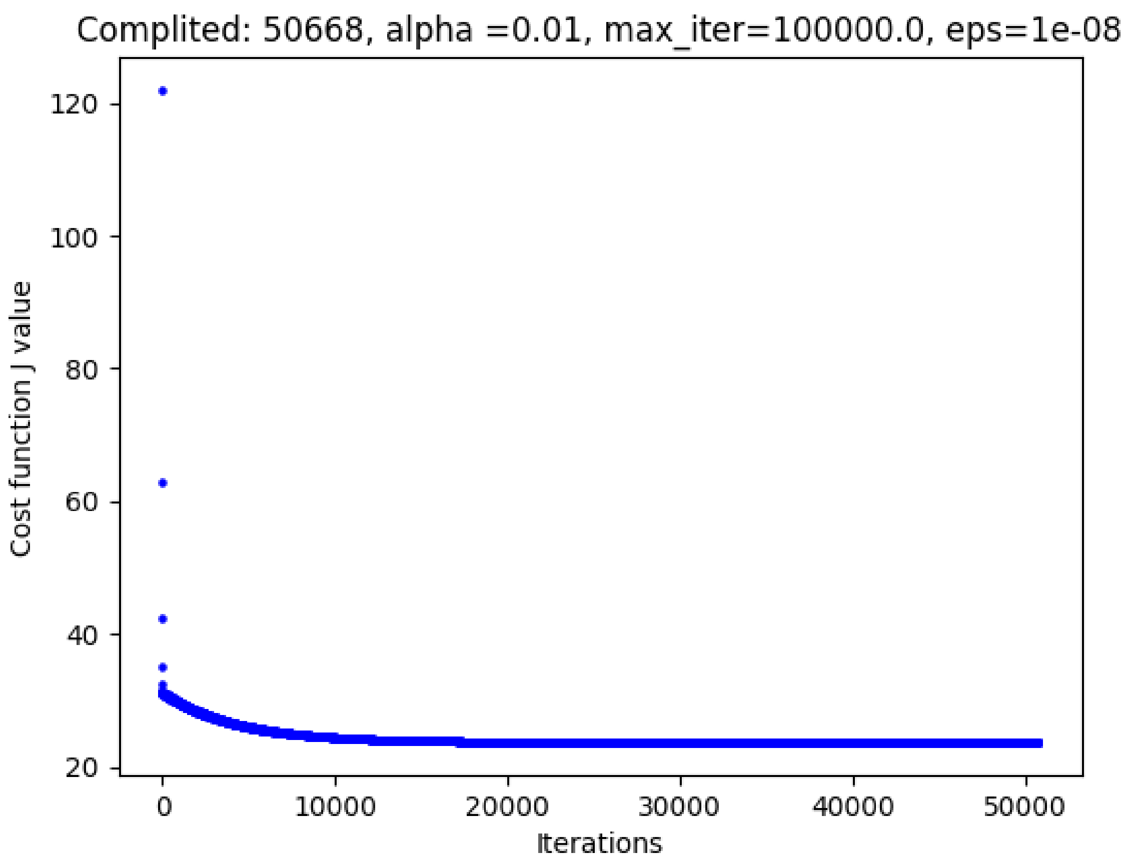
```
        continue_iter = True # flag to continue next iter (grad desc
step)
        iter_number =0 # used for limit by max_iter

        while continue_iter:
            # Do step of gradient descent
            # YOUR_CODE. Insert one step of gradien descent developed
in Linear_Regression_4
            # START_CODE
            dJ_b, dJ_w = self.J_derivative(params, X, y)
            b -= self.alpha * dJ_b
            w -= self.alpha * dJ_w
            params = (b, w)
            # END_CODE

            # keep history of J values
            self.J_hist.append(self.J(self.h(b, w, X), y))
            if self.verbose:
                print ('b = {}, w= {}, J= {}'.format(b,w,self.J_hist[-
1]))
            # check criteria of exit the loop (finish grad desc)
            if self.max_iter and iter_number> self.max_iter: # if
max_iter is provided and limit succeeded
                continue_iter = False
            elif np.abs(self.J_hist[iter_number-1] -
self.J_hist[iter_number])< self.eps: # if accuracy is succeeded
                continue_iter = False
            iter_number += 1

        # store the final params to further using
        self.intercept_, self.coef_= params
        return True

    def draw_cost_changes(self):
        J_hist= self.J_hist[1:]
        plt.figure()

plt.scatter(np.arange(0,len(J_hist)),J_hist,s=20,marker='.',c='b')
        plt.xlabel('Iterations')
        plt.ylabel('Cost function J value')
        title_str = 'Complited: {}, alpha ={}, max_iter={},
eps={}'.format( len(self.J_hist)-2, self.alpha,
self.max_iter,self.eps)
        # Note: len(J_hist)-2) due to first one is -1 (was not
iteration), iter + 1  at the end  of the gradient loop
        plt.title(title_str)


    def predict(self, X):
        '''
```

```
        :param X - ndarray of shape (?,n)
        :return
        '''
        return self.h(self.intercept_, self.coef_, X)


    def score(self, X_test, y_test):
        '''
        :param X_test - ndarray testing set or any for prediction of
shape [?,n], ? - number of samples, n - number of features
        :param y_test - ndarray - 1d array
        :return R2 score of y_test and prediction for X_test
        '''
        z= self.predict(X_test)

        return (r2_score(y_test, z))
```

Check results

```
# DON'T_CHANGE_THIS_CODE. It is used to let you check the result is
correct

print ('X_train.shape= ',X_train.shape)
print ('y_train.shape= ',y_train.shape)
print ('X_train= \n{}'.format (X_train[:5,:]))
lin_reg = Linear_Regression(alpha= 0.01, verbose=0, eps=1e-8)
lin_reg.fit (X_train, y_train)
lin_reg.draw_cost_changes()
print ('R2 Score =', lin_reg.score(X_test, y_test))
print ('b: {}, w= {}'.format(lin_reg.intercept_, lin_reg.coef_))
```

```
X_train.shape=  (379, 1)
y_train.shape=  (379,)
X_train=
[[6.009]
 [5.648]
 [5.885]
 [8.297]
 [6.471]]
R2 Score = 0.5692448312856233
b: -34.31717906874507, w= [[9.01484243]]
```



Expected output:

```
R2 Score = 0.5725111120596516
b: -32.426140228921874, w= [[8.70481894]]
```

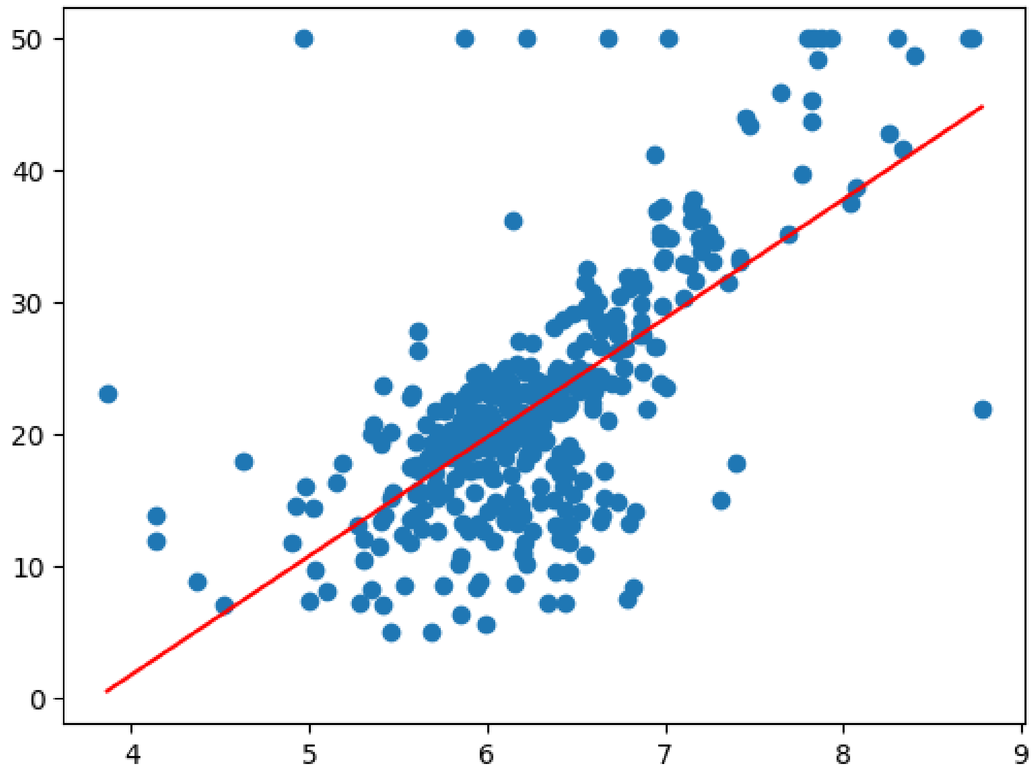
Values differ but not much. This is most likely due to the import of data directly

Draw scatter and prediction for one feature

```
if X_train.shape[1]>1:
    raise Exception ('Select single feature to plot')
plt.figure()
plt.scatter(X_train, y_train)
x_line= np.array([np.min(X_train), np.max(X_train)])
```

```
z_line = lin_reg.predict(x_line.reshape(-1,1))
plt.plot(x_line, z_line, '-', c='red')
```

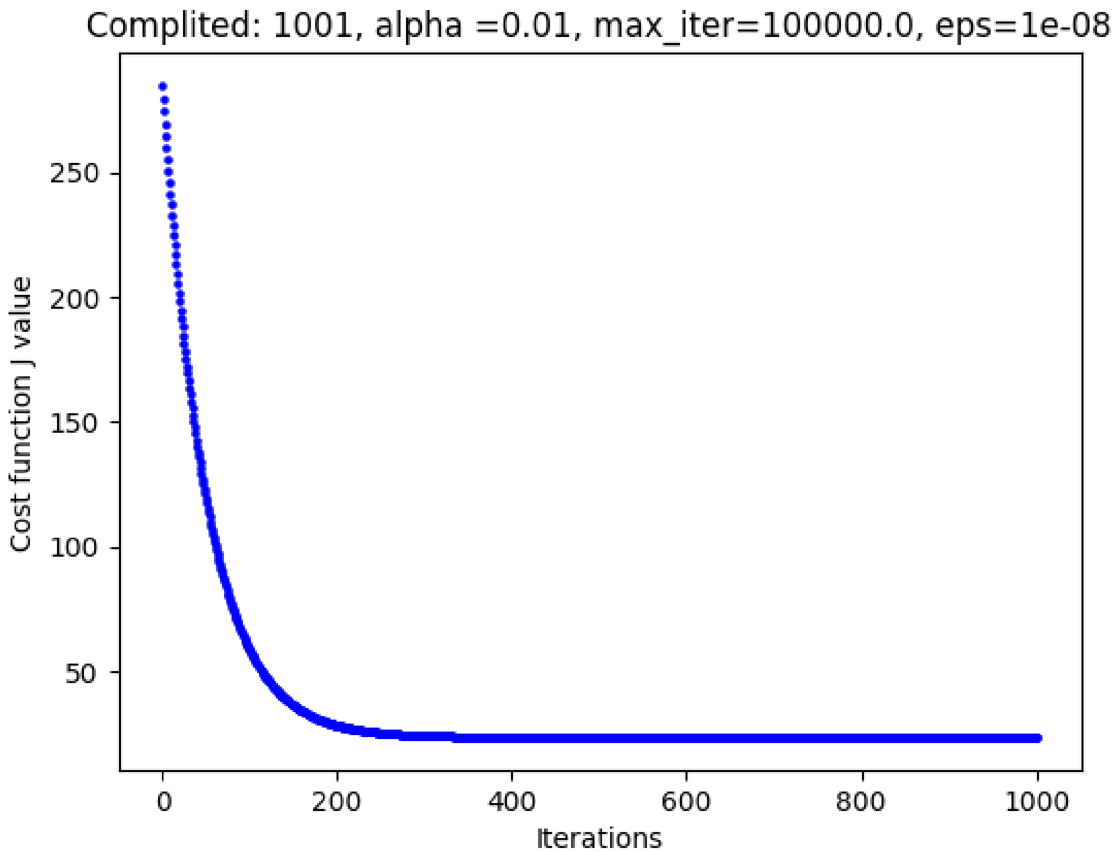
```
[<matplotlib.lines.Line2D at 0x2b9310c3fe0>]
```



Using normalization

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

X_train_scaled= scaler.fit_transform(X_train)
X_test_scaled= scaler.transform(X_test)

lin_reg = Linear_Regression(alpha= 0.01, verbose=0, eps=1e-8)
lin_reg.fit (X_train_scaled, y_train)
print ('R2 Score =',lin_reg.score(X_test_scaled, y_test))
lin_reg.draw_cost_changes()
print ('b: {}, w= {}'.format(lin_reg.intercept_, lin_reg.coef_))
```

```
R2 Score = 0.5692590634340771
b: 22.19853298834398, w= [[6.29700244]]
```

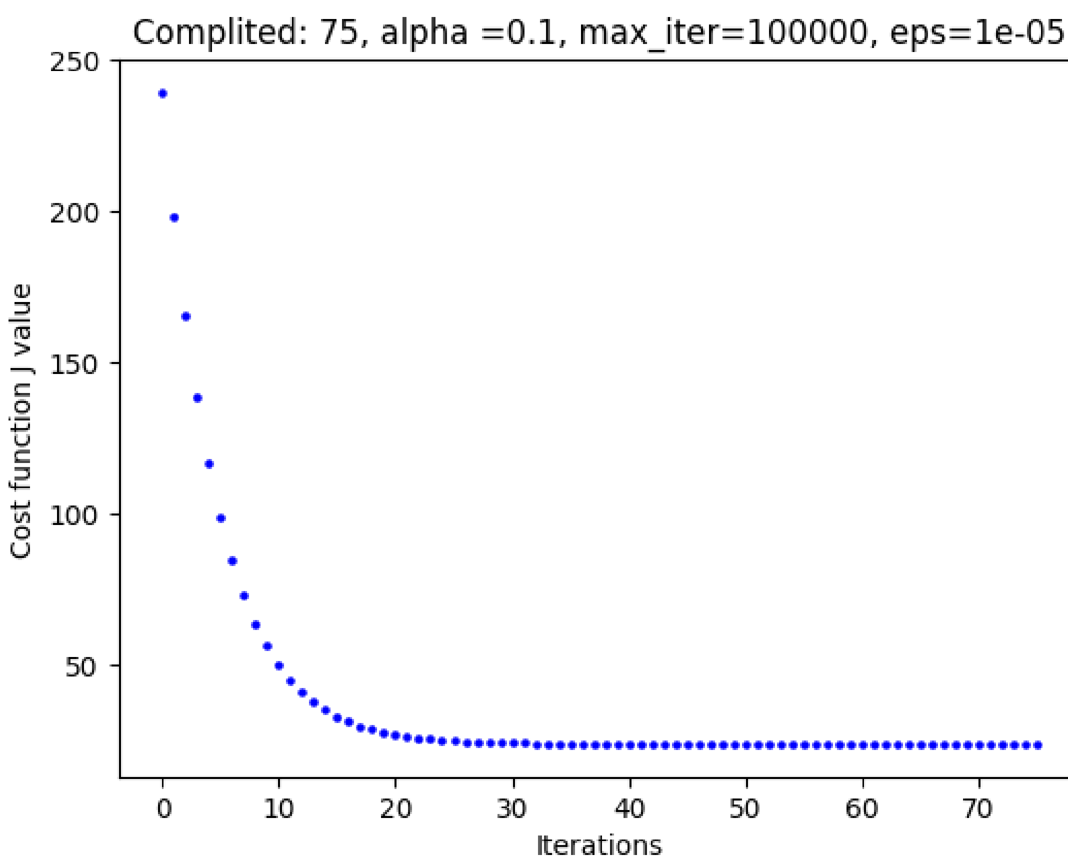


Note: How faster it converges

Run Linear Regression for multi features

```
np.random.seed(2021)
# from sklearn.datasets import load_boston
# X, y = load_boston(return_X_y=True)

# X= X[:,5].reshape(-1,1) # 5 corresponds to 'RM'
# X= X[:,:5]

X_train, X_test, y_train, y_test=  train_test_split(X, y,
random_state=2018)
print ('X_train.shape= ',X_train.shape)
print ('y_train.shape= ',y_train.shape)
X_train[:5]

X_train.shape=  (379, 1)
y_train.shape=  (379,)
```

```
array([[6.009],
       [5.648],
       [5.885],
       [8.297],
       [6.471]])

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_train_scaled= scaler.fit_transform(X_train)
X_test_scaled= scaler.transform(X_test)

lin_reg = Linear_Regression(alpha= 0.1, verbose=0, eps=1e-5,
max_iter=100000)
lin_reg.fit (X_train_scaled, y_train)
lin_reg.draw_cost_changes()
print ('R2 training Score =', lin_reg.score(X_train_scaled, y_train))
print ('R2 Score =', lin_reg.score(X_test_scaled, y_test))
print ('b: {}, w= {}'.format(lin_reg.intercept_, lin_reg.coef_))

R2 training Score = 0.45545273420221755
R2 Score = 0.5691133689561654
b: 22.192080101303628, w= [[6.29517196]]
```



Expected output:

```
R2 training Score = 0.7283111795119549
R2 Score = 0.7714399743645595
b: 22.199472295514532, w= [[-6.71888107e-01  1.10023856e+00
4.11947599e-03  8.26282274e-01
  -2.22625058e+00  2.43471682e+00  2.54149326e-01 -3.29472715e+00
   2.45132782e+00 -1.99309805e+00 -1.95019870e+00  7.67364288e-01
  -4.20581658e+00]]
```

Values differ. This is most likely due to the import of data directly. But it match with SKlearn LinearRegression

Compare with sklearn

```python
from sklearn.linear_model import LinearRegression
lin_reg_sklearn = LinearRegression().fit(X_train_scaled, y_train)
lin_reg_sklearn.score(X_test_scaled, y_test)
```

```
0.5692801665656613
```

Conclusion

The linear regression data that was created in homework and that we import from the sklearn library is the same, I consider the task completed.

The expected output does not match the output of some blocks, in my opinion this is due to the fact that we imported the dataset directly. (not from load_boston())

Although even in the initial task (the file from lesson) there are blocks that do not match with "Expected output"