ep02_linreg_analytic

May 22, 2021

[1]:
```python
name = "Renata Sarmet Smiderle Mendes"  # write YOUR NAME

honorPledge = "I affirm that I have not given or received any unauthorized " \
              "help on this assignment, and that this work is my own.\n"


print("\nName: ", name)
print("\nHonor pledge: ", honorPledge)
```

```
Name:  Renata Sarmet Smiderle Mendes

Honor pledge:  I affirm that I have not given or received any unauthorized help
on this assignment, and that this work is my own.
```

1 MAC0460 / MAC5832 (2021)

2 EP2: Linear regression - analytic solution

2.0.1 Objectives:

- to implement and test the analytic solution for the linear regression task (see, for instance, Slides of Lecture 03 and Lecture 03 of *Learning from Data*)
- to understand the core idea (*optimization of a loss or cost function*) for parameter adjustment in machine learning

3 Linear regression

Given a dataset $\{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(N)}, y^{(N)})\}$ with $\mathbf{x}^{(i)} \in \mathbb{R}^d$ and $y^{(i)} \in \mathbb{R}$, we would like to approximate the unknown function $f : \mathbb{R}^d \to \mathbb{R}$ (recall that $y^{(i)} = f(\mathbf{x}^{(i)})$) by means of a linear model h:

$$h(\mathbf{x}^{(i)}; \mathbf{w}, b) = \mathbf{w}^\top \mathbf{x}^{(i)} + b$$

Note that $h(\mathbf{x}^{(i)}; \mathbf{w}, b)$ is, in fact, an affine transformation of $\mathbf{x}^{(i)}$. As commonly done, we will use the term "linear" to refer to an affine transformation.

The output of h is a linear transformation of $\mathbf{x}^{(i)}$. We use the notation $h(\mathbf{x}^{(i)}; \mathbf{w}, b)$ to make clear that h is a parametric model, i.e., the transformation h is defined by the parameters $\mathbf{w}$ and b. We can view vector $\mathbf{w}$ as a *weight* vector that controls the effect of each *feature* in the prediction.

By adding one component with value equal to 1 to the observations $\mathbf{x}$ (an artificial coordinate), we have:

$$\tilde{\mathbf{x}} = (1, x_1, \ldots, x_d) \in \mathbb{R}^{1+d}$$

and then we can simplify the notation:

$$h(\mathbf{x}^{(i)}; \mathbf{w}) = \hat{y}^{(i)} = \mathbf{w}^\top \tilde{\mathbf{x}}^{(i)}$$

We would like to determine the optimal parameters $\mathbf{w}$ such that prediction $\hat{y}^{(i)}$ is as closest as possible to $y^{(i)}$ according to some error metric. Adopting the *mean square error* as such metric we have the following cost function:

$$J(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \big(\hat{y}^{(i)} - y^{(i)}\big)^2 \tag{1}$$

Thus, the task of determining a function h that is closest to f is reduced to the task of finding the values $\mathbf{w}$ that minimize $J(\mathbf{w})$.

Now we will explore this model, starting with a simple dataset.

3.0.1 Auxiliary functions

[5]:
```
# # Installing libraries
# import sys
# !{sys.executable} -m pip install sklearn
# !{sys.executable} -m pip install pandas
```

[4]:
```
# some imports
import numpy as np
import time
import matplotlib.pyplot as plt

%matplotlib inline
```

[6]:
```
# An auxiliary function
def get_housing_prices_data(N, verbose=True):
    """
    Generates artificial linear data,
    where x = square meter, y = house price

    :param N: data set size
    :type N: int
```

```
    :param verbose: param to control print
    :type verbose: bool
    :return: design matrix, regression targets
    :rtype: np.array, np.array
    """
    cond = False
    while not cond:
        x = np.linspace(90, 1200, N)
        gamma = np.random.normal(30, 10, x.size)
        y = 50 * x + gamma * 400
        x = x.astype("float32")
        x = x.reshape((x.shape[0], 1))
        y = y.astype("float32")
        y = y.reshape((y.shape[0], 1))
        cond = min(y) > 0

    xmean, xsdt, xmax, xmin = np.mean(x), np.std(x), np.max(x), np.min(x)
    ymean, ysdt, ymax, ymin = np.mean(y), np.std(y), np.max(y), np.min(y)
    if verbose:
        print("\nX shape = {}".format(x.shape))
        print("y shape = {}\n".format(y.shape))
        print("X: mean {}, sdt {:.2f}, max {:.2f}, min {:.2f}".format(xmean,
                                                                      xsdt,
                                                                      xmax,
                                                                      xmin))
        print("y: mean {:.2f}, sdt {:.2f}, max {:.2f}, min {:.2f}".format(ymean,
                                                                          ysdt,
                                                                          ymax,
                                                                          ymin))
    return x, y
```

[7]:
```
# Another auxiliary function
def plot_points_regression(x,
                           y,
                           title,
                           xlabel,
                           ylabel,
                           prediction=None,
                           legend=False,
                           r_squared=None,
                           position=(90, 100)):
    """
    Plots the data points and the prediction,
    if there is one.

    :param x: design matrix
    :type x: np.array
```

```
    :param y: regression targets
    :type y: np.array
    :param title: plot's title
    :type title: str
    :param xlabel: x axis label
    :type xlabel: str
    :param ylabel: y axis label
    :type ylabel: str
    :param prediction: model's prediction
    :type prediction: np.array
    :param legend: param to control print legends
    :type legend: bool
    :param r_squared: r^2 value
    :type r_squared: float
    :param position: text position
    :type position: tuple
    """
    fig, ax = plt.subplots(1, 1, figsize=(8, 8))
    line1, = ax.plot(x, y, 'bo', label='Real data')
    if prediction is not None:
        line2, = ax.plot(x, prediction, 'r', label='Predicted data')
        if legend:
            plt.legend(handles=[line1, line2], loc=2)
        ax.set_title(title,
                 fontsize=20,
                 fontweight='bold')
    if r_squared is not None:
        bbox_props = dict(boxstyle="square,pad=0.3",
                          fc="white", ec="black", lw=0.2)
        t = ax.text(position[0], position[1], "$R^2 ={:.4f}$".format(r_squared),
                    size=15, bbox=bbox_props)

    ax.set_xlabel(xlabel, fontsize=20)
    ax.set_ylabel(ylabel, fontsize=20)
    plt.show()
```

3.0.2 The dataset

The first dataset we will use is a toy dataset. We will generate $N = 100$ observations with only one *feature* and a real value associated to each of them. We can view these observations as being pairs *(area of a real state in square meters, price of the real state)*. Our task is to construct a model that is able to predict the price of a real state, given its area.

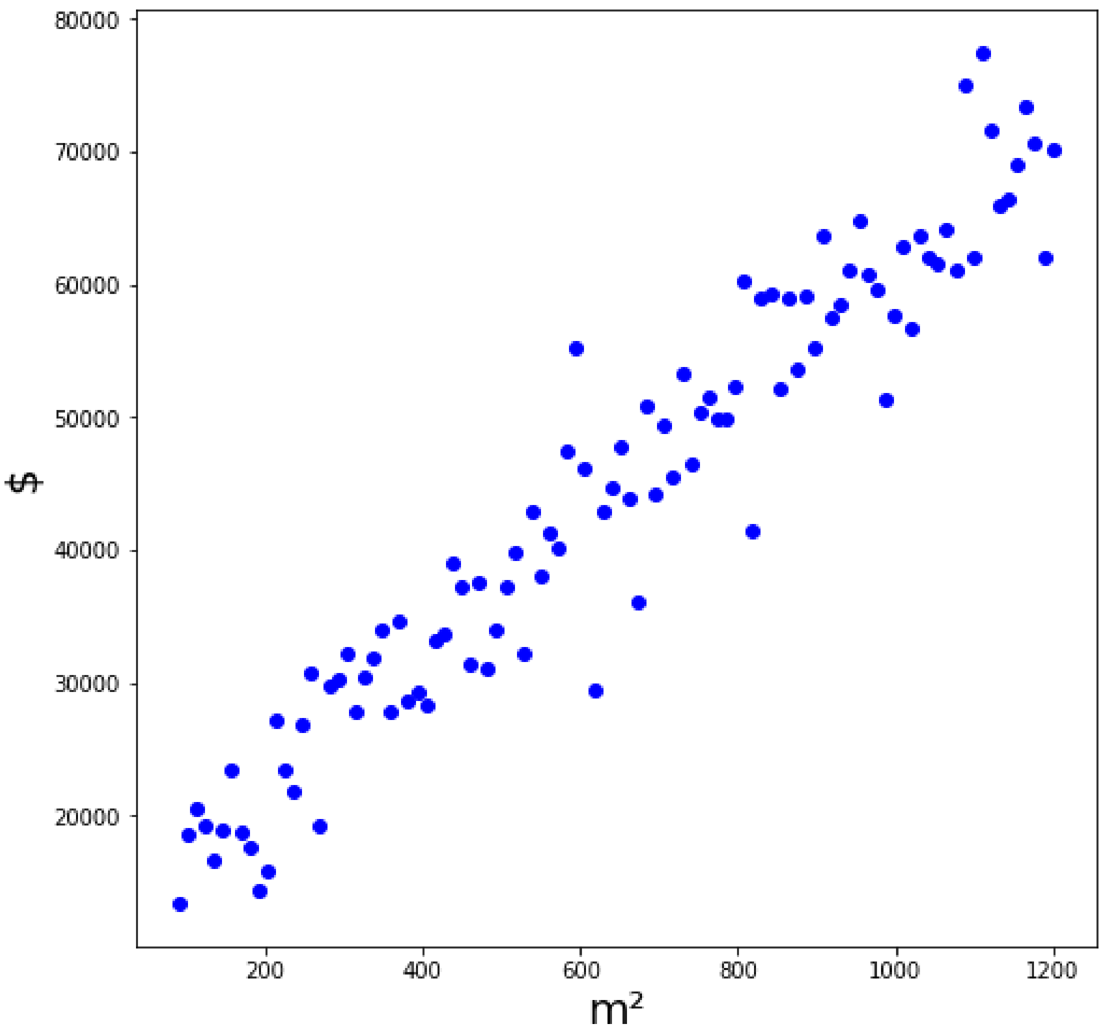
[8]:
```
X, y = get_housing_prices_data(N=100)
```

```
X shape = (100, 1)
y shape = (100, 1)
```

```
X: mean 645.0, sdt 323.65, max 1200.00, min 90.00
y: mean 44278.19, sdt 16521.69, max 77499.13, min 13381.15
```

3.0.3 Ploting the data

[9]:
```python
plot_points_regression(X,
                       y,
                       title='Real estate prices prediction',
                       xlabel="m\u00b2",
                       ylabel='$')
```



3.0.4 The solution

Given $f : \mathbb{R}^{N\times M} \rightarrow \mathbb{R}$ and $\mathbf{A} \in \mathbb{R}^{N\times M}$, we define the gradient of f with respect to $\mathbf{A}$ as:

$$\nabla_{\mathbf{A}} f = \frac{\partial f}{\partial \mathbf{A}} = \begin{bmatrix} \frac{\partial f}{\partial \mathbf{A}_{1,1}} & \cdots & \frac{\partial f}{\partial \mathbf{A}_{1,m}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial \mathbf{A}_{n,1}} & \cdots & \frac{\partial f}{\partial \mathbf{A}_{n,m}} \end{bmatrix}$$

Let $\mathbf{X} \in \mathbb{R}^{N\times d}$ be a matrix (sometimes also called the *design matrix*) whose rows are the observations of the dataset and let $\mathbf{y} \in \mathbb{R}^N$ be the vector consisting of all values $y^{(i)}$ (i.e., $\mathbf{X}^{(i,:)} = \mathbf{x}^{(i)}$ and $\mathbf{y}^{(i)} = y^{(i)}$). It can be verified that:

$$J(\mathbf{w}) = \frac{1}{N}(\mathbf{X}\mathbf{w} - \mathbf{y})^T(\mathbf{X}\mathbf{w} - \mathbf{y}) \tag{2}$$

Using basic matrix derivative concepts we can compute the gradient of $J(\mathbf{w})$ with respect to $\mathbf{w}$:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \frac{2}{N}(\mathbf{X}^T\mathbf{X}\mathbf{w} - \mathbf{X}^T\mathbf{y}) \tag{3}$$

Thus, when $\nabla_{\mathbf{w}} J(\mathbf{w}) = 0$ we have

$$\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\mathbf{y} \tag{4}$$

Hence,

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{5}$$

Note that this solution has a high computational cost. As the number of variables (*features*) increases, the cost for matrix inversion becomes prohibitive. See this text for more details.

4 Exercise 1

Using only **NumPy** (a quick introduction to this library can be found here), complete the two functions below. Recall that $\mathbf{X} \in \mathbb{R}^{N\times d}$; thus you will need to add a component of value 1 to each of the observations in $\mathbf{X}$ before performing the computation described above.

NOTE: Although the dataset above has data of dimension $d = 1$, your code must be generic (it should work for $d \geq 1$)

4.1 1.1. Weight computation function

[10]:
```
def normal_equation_weights(X, y):
    """
    Calculates the weights of a linear function using the normal equation␣
↪method.
    You should add into X a new column with 1s.

    :param X: design matrix
    :type X: np.ndarray(shape=(N, d))
```

```
    :param y: regression targets
    :type y: np.ndarray(shape=(N, 1))
    :return: weight vector
    :rtype: np.ndarray(shape=(d+1, 1))
    """

    # add a left column with 1's into X -- X extended,
    # that way Xi has the same number of elements of the weight array
    Xe = np.hstack((np.ones((X.shape[0],1)), X))

    #   = ( e e)^(-1) e
    w = np.dot(np.dot(np.linalg.inv((np.dot(Xe.T, Xe))),Xe.T), y)

    return w
```

[11]:
```
# test of function normal_equation_weights()

w = normal_equation_weights(X, y)
print("Estimated w =\n", w)
```

```
Estimated w =
 [[12584.29470367]
 [   49.13782612]]
```

4.2 1.2. Prediction function

[12]:
```
def normal_equation_prediction(X, w):
    """
    Calculates the prediction over a set of observations X using the linear␣
 ↪function
    characterized by the weight vector w.
    You should add into X a new column with 1s.

    :param X: design matrix
    :type X: np.ndarray(shape=(N, d))
    :param w: weight vector
    :type w: np.ndarray(shape=(d+1, 1))
    :return y: regression prediction
    :type y: np.ndarray(shape=(N, 1))
    """

    # add a left column with 1's into X -- X extended,
    # that way Xi has the same number of elements of the weight array
    Xe = np.hstack((np.ones((X.shape[0],1)), X))

    # predict
    y = np.dot(Xe,w)
```

```
    return y
```

4.3 1.3. Coefficient of determination

We can use the R^2 metric (Coefficient of determination) to evaluate how well the linear model fits the data.

Which ^{2} value would you expect to observe ?

[13]:
```
from sklearn.metrics import r2_score

# test of function normal_equation_prediction()
prediction = normal_equation_prediction(X, w)

# compute the R2 score using the r2_score function from sklearn
# Replace 0 with an appropriate call of the function
r_2 = r2_score(y, prediction)

plot_points_regression(X,
                       y,
                       title='Real estate prices prediction',
                       xlabel="m\u00b2",
                       ylabel='$',
                       prediction=prediction,
                       legend=True,
                       r_squared=r_2)
```

4.4 Additional tests

Let us compute a prediction for $x = 650$

[14]:
```
# Let us use the prediction function
x = np.asarray([650]).reshape(1,1)
prediction = normal_equation_prediction(x, w)
print("Area = %.2f  Predicted price = %.4f" %(x[0], prediction))
```

```
Area = 650.00  Predicted price = 44523.8817
```

4.5 1.4. Processing time

Experiment with different number of samples N and observe how processing time varies.

Be careful not to use a too large value; it may make jupyter freeze …

[19]:
```
# Add other values for N
N = [100, 1000, 10000, 100000, 1000000, 5000000, 10000000]

for i in N:
    X, y = get_housing_prices_data(N=i)
    init = time.time()
    w = normal_equation_weights(X, y)
    prediction = normal_equation_prediction(X,w)
    init = time.time() - init

    print("\nExecution time = {:.8f}(s)\n".format(init))
```

```
X shape = (100, 1)
y shape = (100, 1)

X: mean 645.0, sdt 323.65, max 1200.00, min 90.00
y: mean 43910.93, sdt 16776.39, max 77778.57, min 15487.14

Execution time = 0.00435472(s)


X shape = (1000, 1)
y shape = (1000, 1)

X: mean 645.0, sdt 320.75, max 1200.00, min 90.00
y: mean 44342.47, sdt 16531.81, max 81215.39, min 8502.54

Execution time = 0.00066280(s)


X shape = (10000, 1)
y shape = (10000, 1)

X: mean 645.0000610351562, sdt 320.46, max 1200.00, min 90.00
y: mean 44215.87, sdt 16500.39, max 82392.02, min 6979.01

Execution time = 0.00505185(s)


X shape = (100000, 1)
y shape = (100000, 1)
```

```
X: mean 645.0000610351562, sdt 320.43, max 1200.00, min 90.00
y: mean 44260.07, sdt 16511.50, max 84762.26, min 3804.07

Execution time = 0.00588608(s)


X shape = (1000000, 1)
y shape = (1000000, 1)

X: mean 645.0000610351562, sdt 320.43, max 1200.00, min 90.00
y: mean 44249.41, sdt 16512.25, max 87467.69, min 1356.34

Execution time = 0.04681587(s)


X shape = (5000000, 1)
y shape = (5000000, 1)

X: mean 644.9999389648438, sdt 320.43, max 1200.00, min 90.00
y: mean 44250.62, sdt 16514.96, max 88842.08, min 270.96

Execution time = 0.16994190(s)


X shape = (10000000, 1)
y shape = (10000000, 1)

X: mean 644.9998779296875, sdt 320.43, max 1200.00, min 90.00
y: mean 44251.12, sdt 16512.59, max 88182.34, min 26.57

Execution time = 0.41862082(s)
```

5 Exercise 2

Let us test the code with > 1. We will use the data we have collected in our first class. The file can be found on e-disciplinas.

Let us try to predict the weight based on one or more features.

[20]:
```python
import pandas as pd

# load the dataset
df = pd.read_csv('QT1data.csv')
df.head()
```

[20]:
```
        Sex  Age  Height  Weight  Shoe number  Trouser number
0    Female   53     154      59           36              40
1      Male   23     170      56           40              38
2    Female   23     167      63           37              40
3      Male   21     178      78           40              40
4    Female   25     153      58           36              38
```

[21]:
```
df.describe()
```

[21]:
```
              Age      Height      Weight  Shoe number
count  130.000000  130.000000  130.000000   130.000000
mean    28.238462  170.684615   70.238462    39.507692
std     12.387042   11.568491   15.534809     2.973386
min      3.000000  100.000000   15.000000    24.000000
25%     21.000000  164.250000   60.000000    38.000000
50%     23.000000  172.000000   69.500000    40.000000
75%     29.000000  178.000000   80.000000    41.000000
max     62.000000  194.000000  130.000000    46.000000
```

[22]:
```
# Our target variable is the weight
y = df.pop('Weight').values
y
```

[22]:
```
array([ 59,  56,  63,  78,  58,  89,  68,  83,  70,  56,  65,  66,  78,
        75,  47,  68,  65,  99,  80,  62,  60,  84,  91,  60,  15,  85,
        56,  62,  69,  78,  60,  48,  66,  85, 101,  74,  52,  52,  80,
        72,  75,  78,  61,  74,  70,  90,  66,  79,  80,  65,  90,  69,
        58,  63,  62,  73,  55,  65,  62,  75,  48,  59,  74,  80,  51,
        90,  58, 117,  77,  75,  56,  50,  67,  93,  70,  76,  85,  50,
        86,  96,  63,  56,  90,  95, 130,  70,  83,  70,  64,  57,  54,
        69,  53,  28,  62,  68,  73,  54,  75,  85,  62,  69,  55,  82,
        84,  52,  64,  73,  86,  77,  64,  65,  55,  50,  98,  77,  51,
        66,  83,  61,  80,  81,  76,  78,  70,  75,  72,  80,  90,  53])
```

5.1 2.1. One feature ($d = 1$)

We will use 'Height' as the input feature and predict the weight

[23]:
```
feature_cols = ['Height']
X = df.loc[:, feature_cols]
X.shape
```

[23]:
```
(130, 1)
```

Write the code for computing the following - compute the regression weights using **X** and **y** - compute the prediction - compute the R^2 value - plot the regression graph (use appropriate values for the parameters of function plot_points_regression())

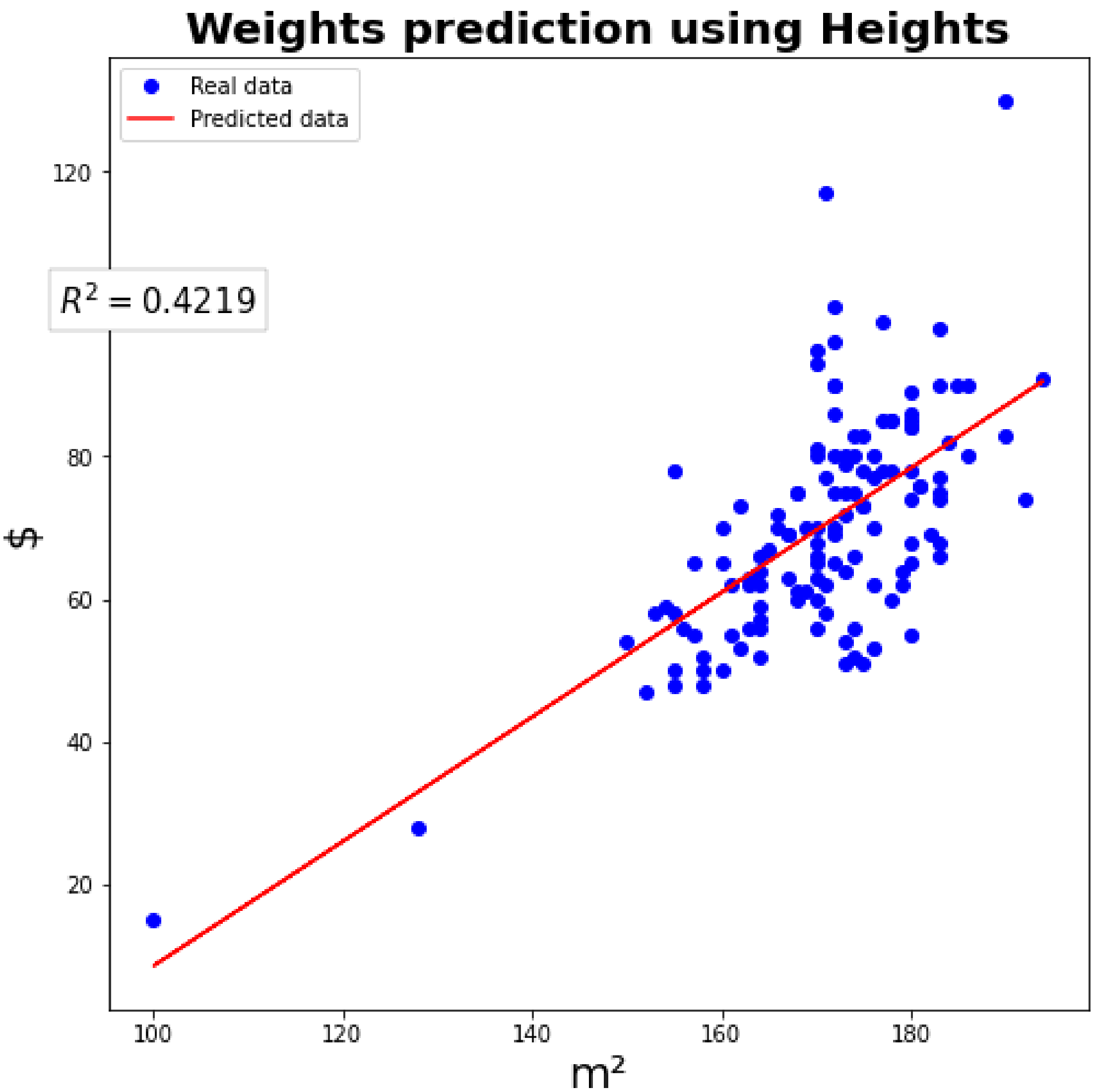
[24]:
```
# Compute the regression weights using X and y
w = normal_equation_weights(X, y)
print("Estimated w =\n", w)

# Compute the prediction
prediction = normal_equation_prediction(X, w)

# Compute the R2 value
r_2 = r2_score(y, prediction)

# Plot the regression graph (use appropriate values for the parameters of␣
↪function plot_points_regression())
plot_points_regression(X,
                       y,
                       title='Weights prediction using Heights',
                       xlabel="m\u00b2",
                       ylabel='$',
                       prediction=prediction,
                       legend=True,
                       r_squared=r_2)
```

```
Estimated w =
 [-78.64309778   0.87226115]
```



5.2 2.2 - Two input features ($d = 2$)

Now repeat the exercise with using as input the features 'Height' and 'Shoe number'

- compute the regression weights using **X** and **y**
- compute the prediction
- compute and print the R^2 value

Note that our plotting function can not be used. There is no need to do plotting here.

[25]:
```
# Select features
feature_cols = ['Height', 'Shoe number']
X = df.loc[:, feature_cols]
print("Shape =", X.shape)
```

```python
# Compute the regression weights using X and y
w = normal_equation_weights(X, y)
print("Estimated w =\n", w)

# Compute the prediction
prediction = normal_equation_prediction(X, w)

# Compute the R2 value
r_2 = r2_score(y, prediction)
print("R2 = {:.4f}".format(r_2))
```

```
Shape = (130, 2)
Estimated w =
 [-80.50372289   0.43049104   1.95566943]
R2 = 0.4538
```

5.3 2.3 - Three input features ($d = 3$)

Now try with three features. There is no need to do plotting here. - compute the regression weights using $\mathbf{X}$ and $\mathbf{y}$ - compute the prediction - compute and print the R^2 value

[26]:
```python
# Select features
feature_cols = ['Height', 'Shoe number', 'Age']
X = df.loc[:, feature_cols]
print("Shape =", X.shape)

# Compute the regression weights using X and y
w = normal_equation_weights(X, y)
print("Estimated w =\n", w)

# Compute the prediction
prediction = normal_equation_prediction(X, w)

# Compute the R2 value
r_2 = r2_score(y, prediction)
print("R2 = {:.4f}".format(r_2))
```

```
Shape = (130, 3)
Estimated w =
 [-87.86115226   0.41007958   2.09107234   0.19448284]
R2 = 0.4776
```

5.4 2.4 - Your comments

Did you observe anything interesting with varying values of d ? Comment about it.

===> Yes, when we increase the value of d, we add more information to it, so the linear model get more explanatory, which is indicated by the larger r2, that is, the better it fits the sample.