Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel→Restart) and then **run all cells** (in the menubar, select Cell→Run All).

Make sure you fill in any place that says `YOUR CODE HERE` or "YOUR ANSWER HERE", as well as your name and collaborators below:

In [1]:
```
NAME = "Nutapol Thungpao"
ID = "st122148"
```

Linear Regression

In this lab, we'll take a look at how to build and evaluate linear regression models. Linear regression works well when there is an (approximately) linear relationship between the features and the variable we're trying to predict.

Before we start, let's import the Python packages we'll need for the tutorial:

In [2]:
```
import matplotlib.pyplot as plt
import numpy as np
```

Univariate example

Here's an example from `[Tim Niven's tutorial at Kaggle](https://www.kaggle.com/timniven/linear-regression-tutorial)`.

Background

We would like to perform *univariate* linear regression using a single feature x, "Number of hours studied," to predict a single dependent variable, y, "Exam score."

We can say that we want to regress `num_hours_studied` onto `exam_score` in order to obtain a model to predict a student's exam score using the number of hours he or she studied.

In the standard setting, we assume that the dependent variable (the exam score) is a random variable that has a Gaussian distribution whose mean is a linear function of the independent variable(s) (the number of hours studied) and whose variance is unknown but constant:

$$y \sim \mathcal{N}(\theta_0 + \theta_1 x, \sigma^2)$$

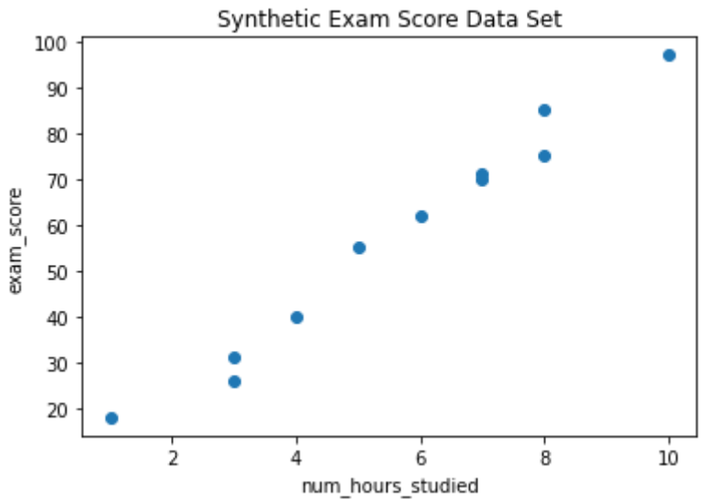
Our model or hypothesis, then, will be a function predicting y based on x:

$$h_\theta(x) = \theta_0 + \theta_1 x$$

Next we'll do something very typical in machine learning experiment: generate some synthetic data for which we know the "correct" model, then use those data to test our algorithm for finding the best model.

So let's generate some example data and examine the relationship between x and y:

In [3]:
```
# Independent variable
num_hours_studied = np.array([1, 3, 3, 4, 5, 6, 7, 7, 8, 8, 10])

# Dependent variable
exam_score = np.array([18, 26, 31, 40, 55, 62, 71, 70, 75, 85, 97])

# Plot the data
plt.scatter(num_hours_studied, exam_score)
plt.xlabel('num_hours_studied')
plt.ylabel('exam_score')
plt.title('Synthetic Exam Score Data Set')
plt.show()
```



Design Matrix

The design matrix, usually written X, contains our independent variables.

In general, with m data points and n features (independent variables), our design matrix will have m rows and n columns.

Note that we have a parameter θ_0, which is the y-intercept term in our linear model. There is no independent variable to multiple θ_0, so we will introduced a dummy variable always equal to 1 to represent the independent variable corresponding to θ_0.

Putting the dummy variable and the number of hours studied together, we obtain the design matrix

$$\mathrm{X} = \begin{bmatrix} 1 & 1 \\ 1 & 3 \\ 1 & 3 \\ 1 & 4 \\ 1 & 5 \\ 1 & 6 \\ 1 & 7 \\ 1 & 7 \\ 1 & 8 \\ 1 & 8 \\ 1 & 10 \end{bmatrix}$$

\ Notice that we do **not** include the dependent variable (exam score) in the design matrix.

In [4]:
```
# Add dummy variable for intercept term to design matrix.
# Understand the numpy insert function by reading https://numpy.org/doc/
stable/reference/generated/numpy.insert.html

X = np.array([num_hours_studied]).T
X = np.insert(X, 0, 1, axis=1)
y = exam_score
print(X.shape)
print(y.shape)
```

```
(11, 2)
(11,)
```

Hypothesis

Let's rewrite the hypothesis function now that we have a dummy variable for the intercept term in the model. We can write the independent variables including the dummy variable as a vector

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix},$$

where $x_0 = 1$ is our dummy variable and x_1 is the number of hours studied. We also write the parameters as a vector

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}.$$

Now we can conveniently write the hypothesis as

$$h_\theta(\mathbf{x}) = \theta^\top \mathbf{x}.$$

Exercise 1 (2 points)

Write a Python code function to evaluate a hypothesis θ for an entire design matrix:

Hint: Use numpy function of `dot`

In [5]:
```
# Evaluate hypothesis over a design matrix
def h(X,theta):
    y_predicted=np.dot(X,theta)#X @ theta
    #raise NotImplementedError()
    return y_predicted
```

In [6]:
```
print(h(X, np.array([0, 10])))
```

```
[ 10  30  30  40  50  60  70  70  80  80 100]
```

Expected output: [10, 30, 30, 40, 50, 60, 70, 70, 80, 80, 100]

Cost function

How can we find the best value of θ? We need a cost function and an algorithm to minimize that cost function.

In a regression problem, we normally use squared error to measure the goodness of fit:

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}\left(h_\theta\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2$$
$$= \frac{1}{2}(\mathrm{X}\theta - \mathbf{y})^\top(\mathrm{X}\theta - \mathbf{y})$$

Here we've used X to denote the design matrix and $\mathbf{y}$ to denote the vector

$$\begin{bmatrix} y_1 \\ \vdots \\ y_m \end{bmatrix}$$

We'll see in a moment how to minimize this cost function.

Exercise 2 (2 points)

Let's implement **cost function** in Python by these steps:

1. Calculate $dy = \hat{y} - y = \mathrm{X}\theta - y$
2. Calcuate $cost = \frac{1}{2}dy^T dy$

In [7]:
```
m = y.shape[0]

def cost(theta, X, y):
    # YOUR CODE HERE
    dy=(np.dot(X,theta)-y)

    J = (1/2)*np.dot(dy,dy)
    #raise NotImplementedError()
    return J
```

In [8]:
```
print(cost(np.array([0, 10]), X, y))
```

```
85.0
```

Expected output: 85.0

Aside: minimizing a convex function using the gradient

To solve our linear regression problem, we want to minimize the cost function $J(\theta)$ above with respect to the parameters θ.

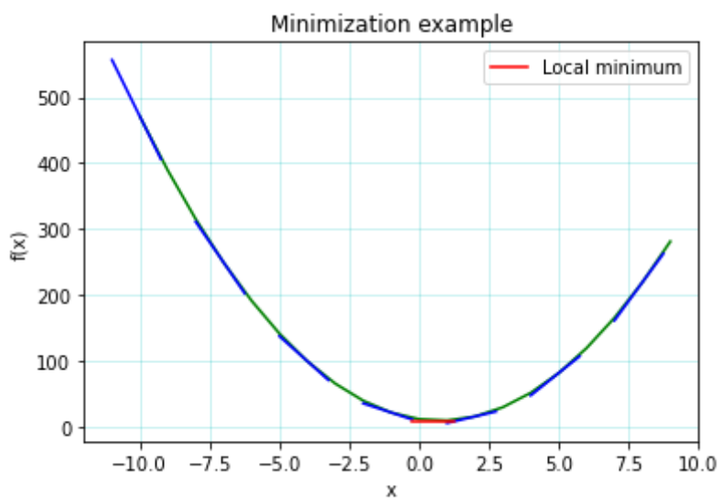
J is convex (see `[Wikipedia](https://en.wikipedia.org/wiki/Convex_function)` for an explanation) so it has just one minimum for some specific value of θ.

To find this minimum, we will find the point at which the gradient is equal to the zero vector.

The gradient of a multivariate function at a particular point is a vector pointing in the direction of maximum slope with a magnitude indicating the slope of the tangent at that point.

To make this clear, let's consider an example in which we consider the function $f(x) = 4x^2 - 6x + 11$ on the interval $[-10, 10]$ and plot its tangent lines at regular intervals.

In [9]:
```
# Define range for plotting x
x = np.arange(-10, 10, 1)

# Example function f(x)
def f(x):
    return 4 * x * x - 6 * x + 11

# Plot f(x)
plt.plot(x, f(x), 'g')

# First derivative of f(x)
def dfx(x):
    return 8 * x - 6

# Plot tangent lines for f(x)
for i in np.arange(-10,10,3):
    x_i = np.arange(i - 1.0, i + 1.0, .25)
    m_i = dfx(i)
    c =  f(i) - m_i*i
    y_i = m_i*(x_i)  +  c
    plt.plot(x_i,y_i,'b')

# Plot tangent line at the minimum of f(x)
minimum = 0.75

for i in [minimum]:
    x_i = np.arange(i - 1, i + 1, .5)
    m_i = dfx(i)
    c = f(i) - m_i * i
    y_i = m_i * (x_i) + c
    plt.plot(x_i, y_i, 'r-', label='Local minimum')

# Decorate the plot
plt.xlabel('x')
plt.ylabel('f(x)')
plt.title('Minimization example')
plt.grid(axis='both',color='c', alpha=0.25)
plt.legend();
plt.show()
```



Minimizing the cost function

Based on the previous example, we can see that to minimize our cost function, we just need to take the gradient with respect to θ and determine where that gradient is equal to $\mathbf{0}$.

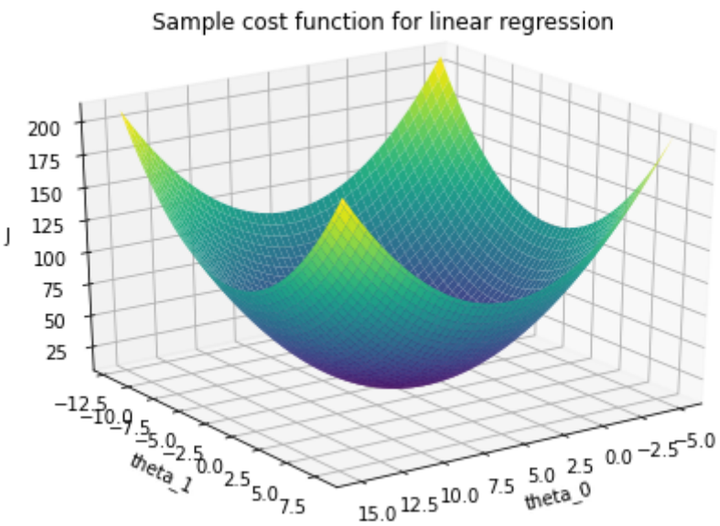
We have

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right)^2.$$

This is a convex function of two variables (θ_0 and θ_1), so it has a single minimum where the gradient $\nabla_J(\theta)$ is $\mathbf{0}$.

Depending on the specific data, the cost function will look something like the surface plotted by the following code. Regardless of where we begin, the gradient always points "uphill," away from the global minimum.

In [10]:
```
# Plot a sample 2D squared error cost function

from mpl_toolkits.mplot3d import Axes3D

x1 = np.linspace(-5.0, 15.0, 100)
x2 = np.linspace(-12.0, 8.0, 100)
X1, X2 = np.meshgrid(x1, x2)
Y = (np.square(X1 - np.mean(X1)) + np.square(X2 - np.mean(X2))) + 10

fig = plt.figure()
ax = Axes3D(fig)
ax.set_xlabel('theta_0')
ax.set_ylabel('theta_1')
ax.set_zlabel('J')
ax.set_title('Sample cost function for linear regression')
cm = plt.cm.get_cmap('viridis')
ax.plot_surface(X1, X2, Y, cmap=cm)
ax.view_init(elev=25, azim=55)
plt.show()
```



Take a look at the lecture notes. If you obtain the partial derivatives of the cost function J with respect to θ, you get

$$\nabla_J(\theta) = \mathrm{X}^\top(\mathrm{X}\theta - \mathbf{y}).$$

Exercise 3 (2 points)

Write the gradient calculation in the equation above as a Python function:

In [11]:
```
# Gradient of cost function

def gradient(X, y, theta):
    # YOUR CODE HERE
    X_theta= np.dot(X,theta)
    grad = np.dot(X.T,(X_theta-y))

    #raise NotImplementedError()
    return grad
```

In [12]:
```
print(gradient(X, y, np.array([0, 10])))
```

```
[-10 -13]
```

Expected output: [-10, -13]

This means that if we currently had the parameter vector [0, 10] (where the cost is 85) and wanted to increase the cost, we could move in the direction [-10, -13]. On the other hand, if we wanted to decrease the cost (which of course we do), we should move in the opposite direction, i.e., [10, 13].

Exercise 4 (2 points)

Implement this idea of gradient descent:

1. Calculate gradient from X, y and θ using function `gradient`
2. Update $\theta_{new} = \theta + \alpha * grad$

In [13]:
```
def gradient_descent(X, y, theta_initial, alpha, num_iters):
    J_per_iter = np.zeros(num_iters)
    gradient_per_iter = np.zeros((num_iters,len(theta_initial)))
    # initialize theta
    theta = theta_initial
    for iter in np.arange(num_iters):
        # YOUR CODE HERE
        grad=gradient(X, y, theta)
        theta = theta - alpha * grad
        #raise NotImplementedError('ValueError')
        J_per_iter[iter] = cost(theta, X, y)
        gradient_per_iter[iter] = grad.T
    return (theta, J_per_iter, gradient_per_iter)
```

In [14]:
```
(theta, J_per_iter, gradient_per_iter) = gradient_descent(X, y, np.array
([0, 10]), 0.001, 10)
print("theta:", theta)
print("J_per_iter:", J_per_iter)
print("gradient_per_iter", gradient_per_iter)
```

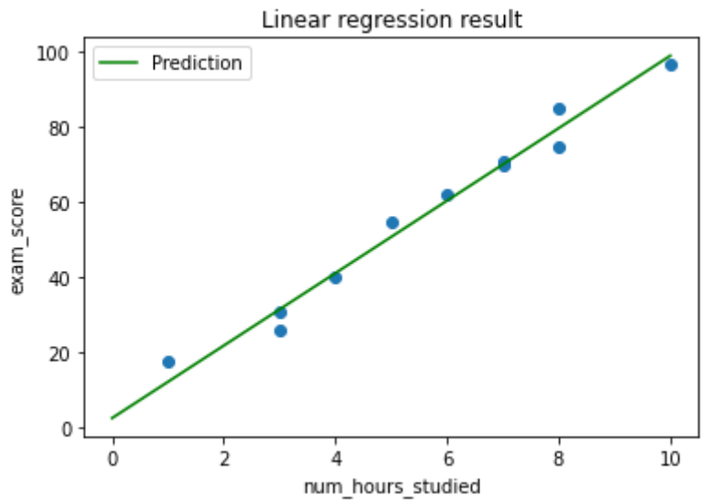
```
theta: [ 0.08327017 10.02116759]
J_per_iter: [84.775269   84.65958757 84.5793525  84.51074587 84.4460598
1 84.38279953
 84.32015717 84.25787073 84.19585485 84.13408132]
gradient_per_iter [[-10.         -13.        ]
 [ -9.084       -6.894     ]
 [ -8.556648    -3.421524  ]
 [ -8.25039038  -1.4471287 ]
 [ -8.06991411  -0.32491618]
 [ -7.96100025   0.31253312]
 [ -7.8928063    0.67422616]
 [ -7.84778746   0.87905671]
 [ -7.81596331   0.9946576 ]
 [ -7.79165648   1.05950182]]
```

Expected output: \ theta: [0.08327017 10.02116759]\ J_per_iter: [84.775269 84.65958757 84.5793525 84.51074587 84.44605981 84.38279953\ 84.32015717 84.25787073 84.19585485 84.13408132]\ gradient_per_iter [[-10. -13.]\ [-9.084 -6.894]\ [-8.556648 -3.421524]\ [-8.25039038 -1.4471287]\ [-8.06991411 -0.32491618]\ [-7.96100025 0.31253312]\ [-7.8928063 0.67422616]\ [-7.84778746 0.87905671]\ [-7.81596331 0.9946576]\ [-7.79165648 1.05950182]]

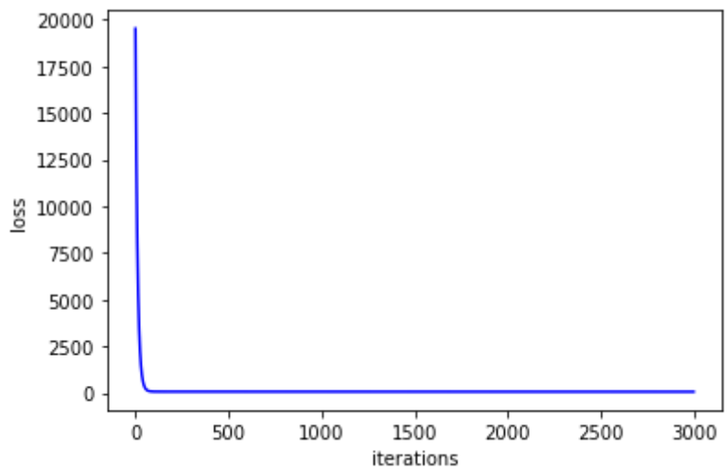
In [15]:
```
# Optimize parameters theta on dataset X, y

theta_initial = np.array([0, 0])
alpha = 0.0001
iterations = 3000
theta, costs, grad = gradient_descent(X, y, theta_initial, alpha, iterat
ions)
print('Optimal parameters: theta_0 %f theta_1 %f' % (theta[0], theta[1
]))
```

```
Optimal parameters: theta_0 2.654577 theta_1 9.641848
```

In [16]:
```
# Visualize the results

plt.scatter(num_hours_studied, exam_score)

x = np.linspace(0,10,20)
y_predicted = theta[0] + theta[1] * x
plt.plot(x, y_predicted, 'g', label='Prediction')

plt.xlabel('num_hours_studied')
plt.ylabel('exam_score')
plt.legend();
plt.title('Linear regression result')
plt.show()
```



In [17]:
```
# Visualize the loss

x_loss = np.arange(0, iterations, 1)

plt.plot(x_loss, costs, 'b-')
plt.xlabel('iterations')
plt.ylabel('loss')
plt.show()
```

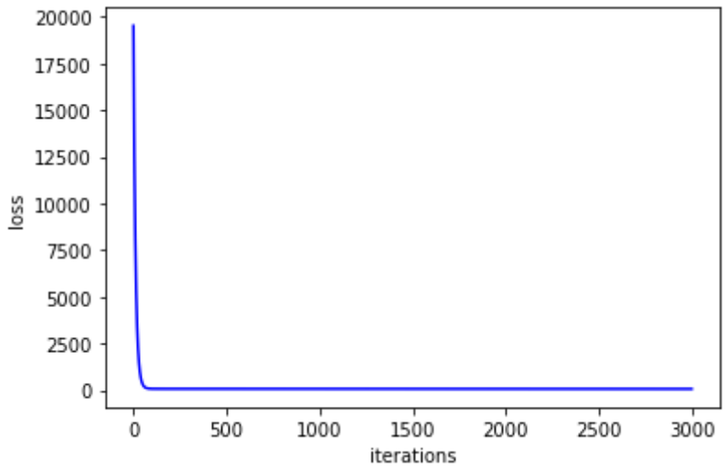


Excercise 5 (2 points)

Instead of repeating the code to plot the loss graph, we would like to encapsulate the code in a function. Complete the loss plotting function below:

In [18]:
```
def cost_plot(iterations, costs):
    x_loss = np.arange(0, iterations, 1)

    plt.plot(x_loss, costs, 'b-')
    plt.xlabel('iterations')
    plt.ylabel('loss')
    plt.show()
```

In [19]:
```
cost_plot(iterations, costs)
```



We can conclude from the loss curve that we have achieved convergence (the loss has stopped improving), and we can conclude that 3000 iterations is overkill! The loss is stable after 100 iterations or so.

Goodness of fit

R^2 is a statistic that will give some information about the goodness of fit of a regression model. The R^2 coefficient of determination is 1 when the regression predictions perfectly fit the data. When R^2 is less than 1, it indicates the percentage of the variance in the target that is accounted for by the prediction.

$$R^2 = 1 - \frac{\sum_{i=1}^{m} \left(y^{(i)} - \hat{y}^{(i)}\right)^2}{\sum_{i=1}^{m} \left(y^{(i)} - \bar{y}^{(i)}\right)^2}$$

Exercise 6 (2 points)

Complete the `goodnees_of_fit` function implementing the equation for R^2 above:

In [20]:
```
def goodness_of_fit(y, y_predicted):
    r_square = 1-((np.sum((y-y_predicted)**2))/(np.sum((y-np.mean(y_pred
icted))**2)))



    return r_square
```

In [21]:
```
y_predicted =  h(X, theta)
r_square = goodness_of_fit(y, y_predicted)
print(r_square)
```

```
0.9786266528258541
```

Expected output: 0.9786239731773175

An R^2 of 0.98 indicates an extremely good (outrageously good, in fact) fit to the data.

Multivariate linear regression example

Next, we extend our model to multiple variables. We'll use a data set from Andrew Ng's class. The data include two independent variables, "Square Feet" and "Number of Bedrooms," and the dependent variable is "Price."

Let's load the data:

In [22]:
```
# We use numpy's genfromtxt function to load the data from the text fil
e.

raw_data = np.genfromtxt('Housing_data.txt',delimiter = ',', dtype=str);

raw_data
```

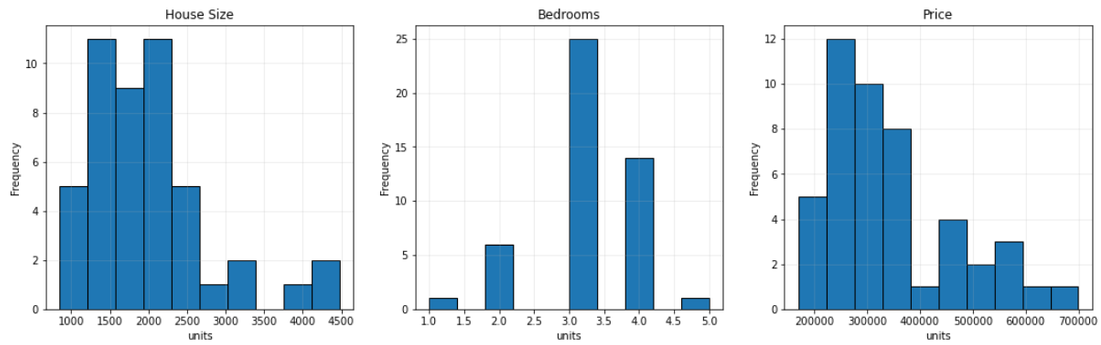
Out[22]:
```
array([['Square Feet', ' Number of bedrooms', 'Price'],
       ['2104', '3', '399900'],
       ['1600', '3', '329900'],
       ['2400', '3', '369000'],
       ['1416', '2', '232000'],
       ['3000', '4', '539900'],
       ['1985', '4', '299900'],
       ['1534', '3', '314900'],
       ['1427', '3', '198999'],
       ['1380', '3', '212000'],
       ['1494', '3', '242500'],
       ['1940', '4', '239999'],
       ['2000', '3', '347000'],
       ['1890', '3', '329999'],
       ['4478', '5', '699900'],
       ['1268', '3', '259900'],
       ['2300', '4', '449900'],
       ['1320', '2', '299900'],
       ['1236', '3', '199900'],
       ['2609', '4', '499998'],
       ['3031', '4', '599000'],
       ['1767', '3', '252900'],
       ['1888', '2', '255000'],
       ['1604', '3', '242900'],
       ['1962', '4', '259900'],
       ['3890', '3', '573900'],
       ['1100', '3', '249900'],
       ['1458', '3', '464500'],
       ['2526', '3', '469000'],
       ['2200', '3', '475000'],
       ['2637', '3', '299900'],
       ['1839', '2', '349900'],
       ['1000', '1', '169900'],
       ['2040', '4', '314900'],
       ['3137', '3', '579900'],
       ['1811', '4', '285900'],
       ['1437', '3', '249900'],
       ['1239', '3', '229900'],
       ['2132', '4', '345000'],
       ['4215', '4', '549000'],
       ['2162', '4', '287000'],
       ['1664', '2', '368500'],
       ['2238', '3', '329900'],
       ['2567', '4', '314000'],
       ['1200', '3', '299000'],
       ['852', '2', '179900'],
       ['1852', '4', '299900'],
       ['1203', '3', '239500']], dtype='<U19')
```

Next, we split the raw data (currently strings) into headers and the data themselves:

In [23]:
```
# Extract headers and data
headers = raw_data[0,:];
print(headers)
data = np.array(raw_data[1:,:], dtype=float);
print(data)
```

```
['Square Feet' ' Number of bedrooms' 'Price']
[[2.10400e+03 3.00000e+00 3.99900e+05]
 [1.60000e+03 3.00000e+00 3.29900e+05]
 [2.40000e+03 3.00000e+00 3.69000e+05]
 [1.41600e+03 2.00000e+00 2.32000e+05]
 [3.00000e+03 4.00000e+00 5.39900e+05]
 [1.98500e+03 4.00000e+00 2.99900e+05]
 [1.53400e+03 3.00000e+00 3.14900e+05]
 [1.42700e+03 3.00000e+00 1.98999e+05]
 [1.38000e+03 3.00000e+00 2.12000e+05]
 [1.49400e+03 3.00000e+00 2.42500e+05]
 [1.94000e+03 4.00000e+00 2.39999e+05]
 [2.00000e+03 3.00000e+00 3.47000e+05]
 [1.89000e+03 3.00000e+00 3.29999e+05]
 [4.47800e+03 5.00000e+00 6.99900e+05]
 [1.26800e+03 3.00000e+00 2.59900e+05]
 [2.30000e+03 4.00000e+00 4.49900e+05]
 [1.32000e+03 2.00000e+00 2.99900e+05]
 [1.23600e+03 3.00000e+00 1.99900e+05]
 [2.60900e+03 4.00000e+00 4.99998e+05]
 [3.03100e+03 4.00000e+00 5.99000e+05]
 [1.76700e+03 3.00000e+00 2.52900e+05]
 [1.88800e+03 2.00000e+00 2.55000e+05]
 [1.60400e+03 3.00000e+00 2.42900e+05]
 [1.96200e+03 4.00000e+00 2.59900e+05]
 [3.89000e+03 3.00000e+00 5.73900e+05]
 [1.10000e+03 3.00000e+00 2.49900e+05]
 [1.45800e+03 3.00000e+00 4.64500e+05]
 [2.52600e+03 3.00000e+00 4.69000e+05]
 [2.20000e+03 3.00000e+00 4.75000e+05]
 [2.63700e+03 3.00000e+00 2.99900e+05]
 [1.83900e+03 2.00000e+00 3.49900e+05]
 [1.00000e+03 1.00000e+00 1.69900e+05]
 [2.04000e+03 4.00000e+00 3.14900e+05]
 [3.13700e+03 3.00000e+00 5.79900e+05]
 [1.81100e+03 4.00000e+00 2.85900e+05]
 [1.43700e+03 3.00000e+00 2.49900e+05]
 [1.23900e+03 3.00000e+00 2.29900e+05]
 [2.13200e+03 4.00000e+00 3.45000e+05]
 [4.21500e+03 4.00000e+00 5.49000e+05]
 [2.16200e+03 4.00000e+00 2.87000e+05]
 [1.66400e+03 2.00000e+00 3.68500e+05]
 [2.23800e+03 3.00000e+00 3.29900e+05]
 [2.56700e+03 4.00000e+00 3.14000e+05]
 [1.20000e+03 3.00000e+00 2.99000e+05]
 [8.52000e+02 2.00000e+00 1.79900e+05]
 [1.85200e+03 4.00000e+00 2.99900e+05]
 [1.20300e+03 3.00000e+00 2.39500e+05]]
```

In [24]:
```
# Visualise the distribution of independent and dependent variables

# Make three subplots, in one row and three columns
fig, ax = plt.subplots(1,3)
fig.set_figheight(5)
fig.set_figwidth(20)
fig.subplots_adjust(left=.2, bottom=None, right=None, top=None, wspace=.
2, hspace=.2)
plt1 = plt.subplot(1,3,1)
plt2 = plt.subplot(1,3,2)
plt3 = plt.subplot(1,3,3)

# Variable 1: square footage
plt1.hist(data[:,0], label='Sq. feet', edgecolor='black')
plt1.set_title('House Size')
plt1.set_xlabel('units')
plt1.set_ylabel('Frequency')
plt1.grid(axis='both', alpha=.25)

# Variable 2: number of bedrooms
plt2.hist(data[:,1], label='Bedroom', edgecolor='black')
plt2.set_title('Bedrooms')
plt2.set_xlabel('units')
plt2.set_ylabel('Frequency')
plt2.grid(axis='both', alpha=.25)

# Variable 3: home price
plt3.hist(data[:,2], label='Price', edgecolor='black')
plt3.set_title('Price')
plt3.set_xlabel('units')
plt3.set_ylabel('Frequency')
plt3.grid(axis='both', alpha=.25)
```



Normalization

We can see from the charts above that the independent variables and the dependent variables have very large differences in their ranges. If you try to use the gradient descent method on these data directly, you may have difficulty in finding a learning rate that is small enough that the costs will not grow out of control but is large enough that the number of iterations is not excessive.

Normalization of the independent and dependent variables can help with this. One type of normalization, sometimes called "standardization" or "z-scaling," involves subtracting a variable's mean then dividing by its standard deviation, calculated over the training samples. The result is a set of standardized variables, each with a mean of 0 and a variance of 1 over the training set.

In [25]:
```
# Normalize the data
means = np.mean(data, axis=0)
stds = np.std(data, axis=0)
data_norm = (data - means) / stds
```

In [26]:
```
# Extract y from the normalized dataset

y_label = 'Price'
y_index = np.where(headers == y_label)[0][0]
y = np.array([data_norm[:,y_index]]).T

# Extract X from normalized dataset

X = data_norm[:,0:y_index]

# Insert column of 1's for intercept term

X = np.insert(X, 0, 1, axis=1)
```

In [27]:
```
# Get number of examples (m) and number of parameters (n)
m = X.shape[0]
n = X.shape[1]
print(m, n)
```

```
47 3
```

Excercise 7 (5 points)

Optimize the parameters using gradient descent:

In [28]:
```
m = y.shape[0]

def cost(theta, X, y):
    # YOUR CODE HERE
    dy=(np.dot(X,theta)-y)

    J = 0.5*np.dot(dy.T,dy)
    #print(J)
    #raise NotImplementedError()
    return J
```

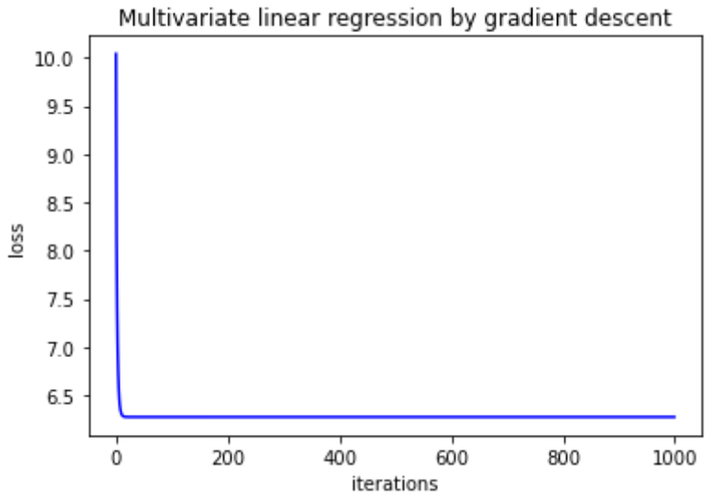
In [29]:
```
theta_initial = np.zeros((X.shape[1],1))
alpha = 0.01
iterations = 1000

# YOUR CODE HERE
# Optimize parameters theta on dataset X, y
def gradient_descent(X, y, theta_initial, alpha, num_iters):
    J_per_iter = np.zeros(num_iters)
    gradient_per_iter = np.zeros((num_iters,len(theta_initial)))
    # initialize theta
    theta = theta_initial
    for iter in np.arange(num_iters):
        # YOUR CODE HERE
        grad=gradient(X, y, theta)
        theta = theta - alpha * grad
        #raise NotImplementedError('ValueError')
        J_per_iter[iter] = cost(theta, X, y)
        gradient_per_iter[iter] = grad.T
    return (theta, J_per_iter, gradient_per_iter)


theta, costs, grad = gradient_descent(X,np.array([data_norm[:,y_index]])
.T, theta_initial, alpha, iterations)
```

In [30]:
```
print('Theta values ', theta)
```

```
Theta values  [[-9.79771819e-17]
 [ 8.84765988e-01]
 [-5.31788197e-02]]
```

Expected output:\ Theta values [[-9.15933995e-17]\ [8.84765988e-01]\ [-5.31788197e-02]]

In [31]:
```
# Visualize the loss over the optimization
plt.title('Multivariate linear regression by gradient descent')
cost_plot(iterations, costs)
```



Transforming parameters back to the original scale Now that we've got optimal parameters for our original data, we need to undo the normalization.

We have

$$\hat{y}^{\text{norm}} = \theta^{\text{norm}} \mathbf{x}^{\text{norm}}$$

Excercise 8 (3 points)

Modify the code to compute goodness of fit

In [32]:
```
# Goodness of fit
y_predicted = h(X,theta)
# YOUR CODE HERE
r_square = goodness_of_fit(y,y_predicted)
#raise NotImplementedError()
```

In [33]:
```
print(r_square)
```

```
0.7329450180289143
```

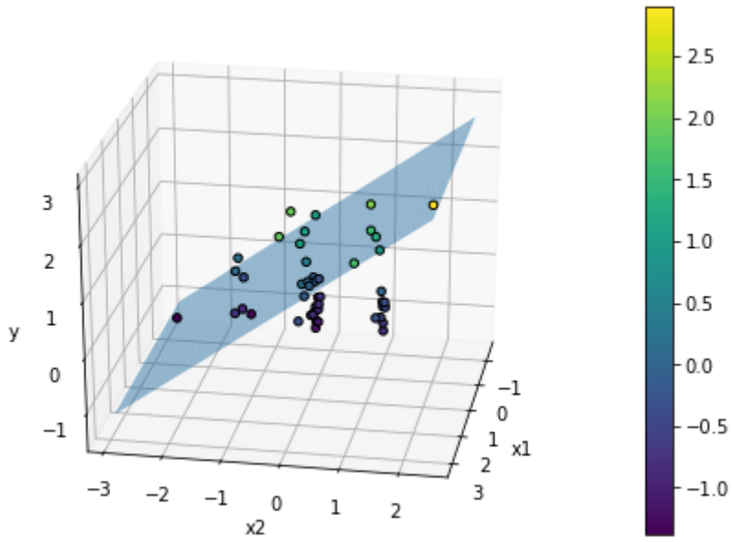
Transform standardized data back to original scale

We can transform standardized predicted values, y*predicted into the orginal data scale using$$y*{\text{norm}} = \sigma_y y + \mu_y$$

In [34]:
```
# Compute mean and standard deviation of data

sigma = np.array(np.std(data,axis=0))
mu = np.array(np.mean(data,axis=0))

# De-normalize y

y_predicted =  np.round(h(X, theta) * sigma[2] + mu[2])

# Print first five values of y_predicted

print(y_predicted[0:5,:])
```

```
[[356283.]
 [286121.]
 [397489.]
 [269244.]
 [472278.]]
```

In [35]:
```
# 3D plot of standardized data

from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = Axes3D(fig)
p = ax.scatter(X[:,1],X[:,2],y,edgecolors='black',c=data_norm[:,2],alpha
=1)
ax.set_xlabel('x1')
ax.set_ylabel('x2')
ax.set_zlabel('y')

X1 = np.linspace(min(X[:,1]), max(X[:,1]), len(y))
X2 = np.linspace(min(X[:,2]), max(X[:,2]), len(y))

xx1,xx2 = np.meshgrid(X1,X2)

yy = (theta[0] + theta[1]*xx1.T + theta[2]*xx2)
ax.plot_surface(xx1,xx2,yy, alpha=0.5)
ax.view_init(elev=25, azim=10)
plt.colorbar(p)
plt.show()
```



In-class exercises

Now that you're familiar with minimizing a cost function using its gradient and gradient descent, refer to the lecture notes to find the analytical solution (the normal equations) to the linear regression problem.

Implement the normal equation approach for the synthetic univariate data set and the housing price data set. Demonstrate your solution in the lab.

In [36]:
```
# just remove all parameters
%reset
```

In [37]:
```
import matplotlib.pyplot as plt
import numpy as np
```

Exercise 2.1 (5 points)

Download raw_data and setup data

In [38]:
```
# Download raw_data and setup data
# YOUR CODE HERE
data = np.genfromtxt('Housing_data.txt',delimiter = ',', dtype=str,);
headers = data[0,:];
data = np.array(data[1:,:], dtype=float);


#raise NotImplementedError()
```

In [39]:
```
print(data[:5])
```

```
[[2.104e+03 3.000e+00 3.999e+05]
 [1.600e+03 3.000e+00 3.299e+05]
 [2.400e+03 3.000e+00 3.690e+05]
 [1.416e+03 2.000e+00 2.320e+05]
 [3.000e+03 4.000e+00 5.399e+05]]
```

Expected result:\ [[2.104e+03 3.000e+00 3.999e+05]\ [1.600e+03 3.000e+00 3.299e+05]\ [2.400e+03 3.000e+00 3.690e+05]\ [1.416e+03 2.000e+00 2.320e+05]\ [3.000e+03 4.000e+00 5.399e+05]]

Exercise 2.2 (5 points)

Normalized data

In [40]:
```
# Normalized data
def normalized_data(data):
    # YOUR CODE HERE
    means = np.mean(data, axis=0)
    stds = np.std(data, axis=0)
    data_norm = (data - means) / stds
    return data_norm
```

In [41]:
```
data_norm = normalized_data(data)
print(data_norm[:5])
```

```
[[ 0.13141542 -0.22609337  0.48089023]
 [-0.5096407  -0.22609337 -0.08498338]
 [ 0.5079087  -0.22609337  0.23109745]
 [-0.74367706 -1.5543919  -0.87639804]
 [ 1.27107075  1.10220517  1.61263744]]
```

Expected result:\ [[0.13141542 -0.22609337 0.48089023]\ [-0.5096407 -0.22609337 -0.08498338]\ [0.5079087 -0.22609337 0.23109745]\ [-0.74367706 -1.5543919 -0.87639804]\ [1.27107075 1.10220517 1.61263744]]

Exercise 2.3 (5 points)

Extract X and y from data

In [42]:
```
# Extract y from the normalized dataset
y_label = 'Price'
y_index = np.where(headers == y_label)[0][0]
y = np.array([data_norm[:,y_index]]).T
```

In [43]:
```
print(y[:5])
```

```
[[ 0.48089023]
 [-0.08498338]
 [ 0.23109745]
 [-0.87639804]
 [ 1.61263744]]
```

Expected result: [0.48089023 -0.08498338 0.23109745 -0.87639804 1.61263744]

In [44]:
```
# Extract X from data
X = data_norm[:,0:y_index]
X = np.insert(X, 0, 1, axis=1)
```

In [45]:
```
print(X[:5,:])
```

```
[[ 1.          0.13141542 -0.22609337]
 [ 1.         -0.5096407  -0.22609337]
 [ 1.          0.5079087  -0.22609337]
 [ 1.         -0.74367706 -1.5543919 ]
 [ 1.          1.27107075  1.10220517]]
```

Expected result:\ [[1. 0.13141542 -0.22609337]\ [1. -0.5096407 -0.22609337]\ [1. 0.5079087 -0.22609337]\ [1. -0.74367706 -1.5543919]\ [1. 1.27107075 1.10220517]]

Exercise 2.4 (8 points)

Create h, cost, gradient, and gradient_descent

In [46]:
```
# create h function
def h(X,theta):
    y_pre=np.dot(X,theta)
    return y_pre
```

In [47]:
```
print(h(X, np.array([1, 2, 4]))[:5])
```

```
[ 0.35845737 -0.92365487  1.11144393 -6.70492173  7.95096216]
```

Expected result: [0.35845737 -0.92365487 1.11144393 -6.70492173 7.95096216]

In [48]:
```
#def cost(theta, X, y):
    # YOUR CODE HERE
 #   raise NotImplementedError()
  #  return J
m=y.shape[0]

def cost(theta, X, y):
    y=y.reshape(47,)
    dy=h(X,theta)-y
    J = 0.5*np.dot(dy.T,dy)
    return J
```

In [49]:
```
print(cost(np.array([1, 8, 10]), X, y))
```

```
5477.13862837469
```

Expected result: 5477.138628374691

In [50]:
```
# Gradient of cost function
def gradient(X, y, theta):
    y=y.reshape(47,)
    dy=h(X,theta)-y
    X_theta= np.dot(X,theta)
    grad = np.dot(X.T,(dy))


    return grad
```

In [51]:
```
print(gradient(X, y, np.array([1, 8, 10])))
```

```
[ 47.         599.00016917 659.76139633]
```

Expected result: [47. 599.00016917 659.76139633]

In [52]:
```
def gradient_descent(X, y, theta_initial, alpha, num_iters):
    J_per_iter = np.zeros(num_iters)
    gradient_per_iter = np.zeros((num_iters,len(theta_initial)))
    # initialize theta
    theta = theta_initial
    loss=[]
    for iter in np.arange(num_iters):
        # YOUR CODE HERE
        grad=gradient(X, y, theta)
        theta = theta - alpha * grad
        costs=cost(theta, X, y)
        J_per_iter[iter] = cost(theta, X, y)
        gradient_per_iter[iter] = grad.T
        loss.append(cost)
    return (theta, J_per_iter, gradient_per_iter)
```

In [53]:
```
(theta, J_per_iter, gradient_per_iter) = gradient_descent(X, y, np.array
([0, 1, 10]), 0.001, 10)
print("theta:", theta)
print("J_per_iter:", J_per_iter)
print("gradient_per_iter", gradient_per_iter)
```

```
theta: [-8.28226376e-16 -7.72838948e-01  6.35294636e+00]
J_per_iter: [2123.51284628 1873.56259758 1656.90935568 1468.93187452 13
05.65834104
 1163.67477334 1040.04635308  932.24986509  838.11567544  755.77790087]
gradient_per_iter [[1.31450406e-13 2.70000169e+02 4.75532186e+02]
 [1.05693232e-13 2.44794887e+02 4.46076185e+02]
 [1.07025500e-13 2.21549490e+02 4.18667980e+02]
 [8.74855743e-14 2.00117968e+02 3.93159744e+02]
 [1.03916875e-13 1.80365065e+02 3.69414440e+02]
 [7.04991621e-14 1.62165488e+02 3.47305031e+02]
 [6.29496455e-14 1.45403177e+02 3.26713748e+02]
 [5.29576383e-14 1.29970626e+02 3.07531415e+02]
 [5.98410210e-14 1.15768253e+02 2.89656812e+02]
 [4.64073224e-14 1.02703825e+02 2.72996100e+02]]
```

Expected result: theta: [-8.20787882e-16 -7.72838948e-01 6.35294636e+00]\ J_per_iter: [2123.51284628 1873.56259758 1656.90935568 1468.93187452 1305.65834104\ 1163.67477334 1040.04635308 932.24986509 838.11567544 755.77790087]\ gradient_per_iter [[1.31450406e-13 2.70000169e+02 4.75532186e+02]\ [9.68114477e-14 2.44794887e+02 4.46076185e+02]\ [9.63673585e-14 2.21549490e+02 4.18667980e+02]\ [8.92619312e-14 2.00117968e+02 3.93159744e+02]\ [1.11022302e-13 1.80365065e+02 3.69414440e+02]\ [7.40518757e-14 1.62165488e+02 3.47305031e+02]\ [5.05151476e-14 1.45403177e+02 3.26713748e+02]\ [6.09512441e-14 1.29970626e+02 3.07531415e+02]\ [6.29496455e-14 1.15768253e+02 2.89656812e+02]\ [4.74065232e-14 1.02703825e+02 2.72996100e+02]]\

Exercise 2.5 (5 points)

Do optimization using gradient descent with $\alpha = 0.003$ and 30,000 iterations

In [54]:
```
(theta, costs, grad) = gradient_descent(X, y, np.array([0, 1, 10]), 0.00
3, 30000)
```

In [55]:
```
print("theta:", theta)
print("cost_per_iter:", costs[-5:])
print("gradient_per_iter", grad[-5:])
```

```
theta: [-7.66053887e-17  8.84765988e-01 -5.31788197e-02]
cost_per_iter: [6.27579208 6.27579208 6.27579208 6.27579208 6.27579208]
gradient_per_iter [[ 3.05311332e-16 -1.78468351e-14  1.15185639e-15]
 [-1.38777878e-16 -1.78468351e-14  1.15185639e-15]
 [-1.38777878e-16 -1.78468351e-14  1.15185639e-15]
 [ 3.05311332e-16 -1.78468351e-14  1.15185639e-15]
 [-1.38777878e-16 -1.78468351e-14  1.15185639e-15]]
```

Expected result:\ theta: [-1.05832010e-16 8.84765988e-01 -5.31788197e-02]\ J_per_iter: [6.27579208 6.27579208 6.27579208 6.27579208 6.27579208]\ gradient_per_iter [[0.00000000e+00 -1.72082220e-14 8.75724041e-16]\ [0.00000000e+00 -1.72082220e-14 8.75724041e-16]\ [0.00000000e+00 -1.72082220e-14 8.75724041e-16]\ [0.00000000e+00 -1.72082220e-14 8.75724041e-16]\ [0.00000000e+00 -1.72082220e-14 8.75724041e-16]]

Exercise 2.6 (2 points)

Calculate goodness of fit

In [56]:
```
def goodness_of_fit(y, y_predicted):
    #print(np.shape(y))
    y=y.reshape(np.shape(y)[0],)
    #up=np.sum((y-y_predicted))

    #low=np.sum((y-y.mean()))

    r_square = 1-((np.sum((y-y_predicted)**2))/(np.sum((y-np.mean(y))**2
)))
    #r_square=(up/low)
    return r_square
```

In [57]:
```
y_predicted =  h(X, theta)
r_square = goodness_of_fit(y, y_predicted)
print(r_square)
```

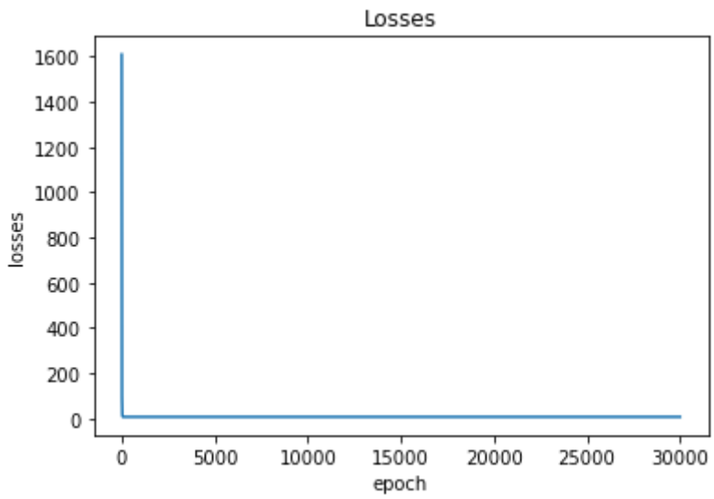
```
0.7329450180289143
```

Expected result: 0.7329450180289143

Excercise 2.7 (2 point)

Plot graph of cost results

In [58]:
```
def gradient_descent(X, y, theta_initial, alpha, num_iters):
    J_per_iter = np.zeros(num_iters)
    gradient_per_iter = np.zeros((num_iters,len(theta_initial)))
    # initialize theta
    theta = theta_initial
    loss=[]
    for iter in np.arange(num_iters):
        # YOUR CODE HERE
        grad=gradient(X, y, theta)
        theta = theta - alpha * grad
        costs=cost(theta, X, y)
        J_per_iter[iter] = cost(theta, X, y)
        gradient_per_iter[iter] = grad.T
        loss.append(costs)
    cost_plot(loss)
    return (theta, J_per_iter, gradient_per_iter)

def cost_plot(loss):
    #def cost_plot(iterations, costs):
    plt.plot(np.arange(len(loss)),loss,label= ' Train losses')
    plt.title('Losses')
    plt.xlabel('epoch')
    plt.ylabel('losses')
(theta, costs, grad) = gradient_descent(X, y, np.array([0, 1, 10]), 0.00
3, 30000)
```



Exercise 2.8 (8 points)

Write a function implementing the normal equations for linear equation:

In [59]:
```
def goodness_of_fit(y, y_predicted):

    y_predicted=y_predicted.reshape(np.shape(y))
    r_square = 1-((np.sum((y-y_predicted)**2))/(np.sum((y-np.mean(y_pred
icted))**2)))
    #print(np.shape(y))
    return r_square
```

In [60]:
```
from numpy.linalg import inv
def normal_equation(X, y):
    num_iters=30000000
    theta_initial=np.array([0, 1, X.shape[0]])
    J_per_iter = np.zeros(num_iters)
    gradient_per_iter = np.zeros((num_iters,len(theta_initial)))
    alpha = 0.000001
    # initialize theta
    #theta = theta_initial
    X_dot= np.dot(X.T,X)
    X_dot_inv=np.linalg.inv(X_dot)
    X_dot_y=X.T@y
    theta =X_dot_inv @X_dot_y
    #for iter in np.arange(num_iters):
        # YOUR CODE HERE
        #grad=gradient(X, y, theta)

        #costs=cost(theta, X, y)
        #J_per_iter[iter] = cost(theta, X, y)
        #gradient_per_iter[iter] = grad.T
    #return (theta, J_per_iter, gradient_per_iter)
    return theta.T
```

In [61]:
```
theta_norm = normal_equation(X,np.array([y]).T)
print("theta from normal equation:", theta_norm.T)
y_norm_predicted =  h(X, theta_norm)
r_norm_square = goodness_of_fit(y, y_norm_predicted)
print("r_square:", r_norm_square)
```

```
theta from normal equation: [[[-7.76616596e-17]
  [ 8.84765988e-01]
  [-5.31788197e-02]]]
r_square: 0.7329450180289143
```

Expected result:\ theta from normal equation: [[-7.90434550e-17 8.84765988e-01 -5.31788197e-02]]\ r_square: 0.7329450180289143

Take-home exercise (40 points)

Find an interesting dataset for linear regression on Kaggle. Implement the normal equations and gradient descent then evaluate your model's performance.

Write a brief report on your experiments and results in the form of a Jupyter notebook.

Explain the dataset which you get and which rows which you use. How many data in your dataset?

```
Datasets that,I am interesting is Average age people in a country lived.(htt
ps://www.kaggle.com/brendan45774/countries-life-expectancy) The data collect
for 15 country from 1800 to 2016. Ann I use data 1900 to 2016 for calculatio
n. That have 117 rows × 2 columns by columns of Year that get the data and L
ife expectancy. For result of gradient descent  is  Theta values  [[0.012962
17][0.47441071][0.48764956]] ,r_square :0.9826781891092137 . And For the res
ult of normal equations is theta from normal equation: [[-1.30104261e-16][-
7.10542736e-15][ 1.00000000e+00]] r_square: 1.0
```

Write down your all code at below. Show the results, goodness of fit and plot cost graph

In [1]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns

from sklearn.datasets import load_boston
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
import numpy as np
from time import time
import math,random
from numpy.random import default_rng
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report
import matplotlib.pyplot as plt
```

Datasets that,I am interesting is Average age people in a country lived. The data collect for 15 country from 1800 to 2016. And I use data 1900 to 2016 f or calculation. That have 117 rows × 2 columns by columns of Year that get t he data and Life expectancy.

1) Step import data from file csv by Pandas

In [12]:
```
data_raw = pd.read_csv('Life expectancy.csv')
df=pd.DataFrame(data_raw)
```

2) Print information of data.

In [13]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3253 entries, 0 to 3252
Data columns (total 3 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   Entity           3253 non-null   object 
 1   Year             3253 non-null   int64  
 2   Life expectancy  3253 non-null   float64
dtypes: float64(1), int64(1), object(1)
memory usage: 76.4+ KB
```

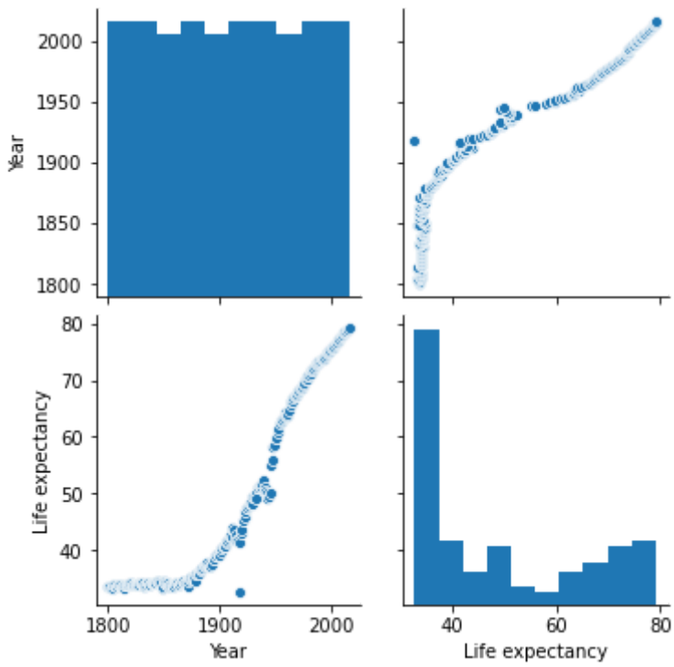
In [14]:
```
df.isnull().sum()
```

Out[14]:
```
Entity             0
Year               0
Life expectancy    0
dtype: int64
```

In [15]:
```
df=df.groupby('Year').mean().reset_index()  #I use mean of Life expectan
cy that mean of Life expectancy of people of the world
```

In [16]:
```
sns.pairplot(df)
```

Out[16]:
```
<seaborn.axisgrid.PairGrid at 0x7f41ce4967f0>
```

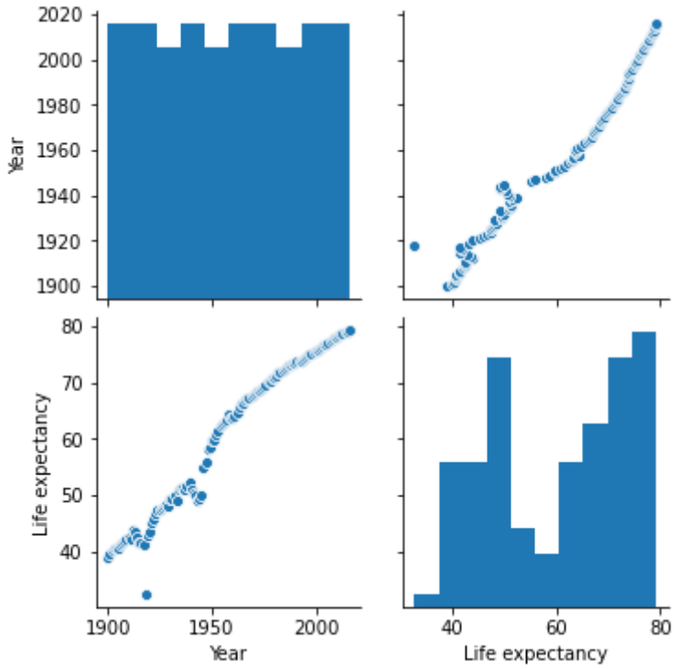


I cut data from 1800-1899 out because many country Life expectancy is constant for long time that almost do not change for 100 year.

In [17]:
```
df=df[df['Year']>=1900]
```

In [18]:
```
sns.pairplot(df)
```

Out[18]:
```
<seaborn.axisgrid.PairGrid at 0x7f41cdf9bee0>
```



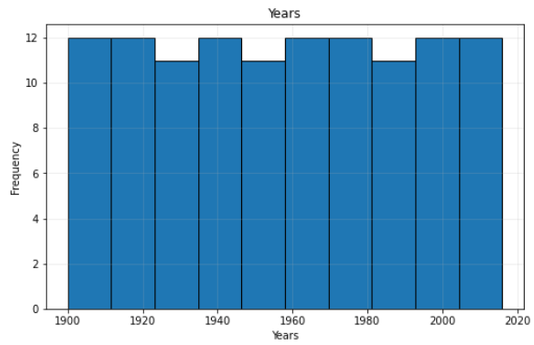
I change name of country to number 1 to 15

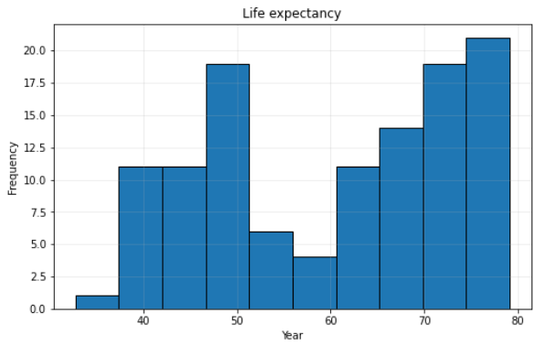
In [19]:
```
data=df.to_numpy()    #import data from pandas to numpy
```

In [34]:
```
data.shape
```

Out[34]:
```
(117, 2)
```

In [23]:
```
# Visualise the distribution of independent and dependent variables

fig, ax = plt.subplots(1,2)
fig.set_figheight(5)
fig.set_figwidth(20)
fig.subplots_adjust(left=.2, bottom=None, right=None, top=None, wspace=.
2, hspace=.2)
plt1 = plt.subplot(1,2,1)
plt2 = plt.subplot(1,2,2)


# Years
plt1.hist(data[:,0], label='Years', edgecolor='black')
plt1.set_title('Years')
plt1.set_xlabel('Years')
plt1.set_ylabel('Frequency')
plt1.grid(axis='both', alpha=.25)

# Variable 2: Life expectancy
plt2.hist(data[:,1], label='Life expectancy', edgecolor='black')
plt2.set_title('Life expectancy')
plt2.set_xlabel('Year')
plt2.set_ylabel('Frequency')
plt2.grid(axis='both', alpha=.25)
```





Normalization

In [24]:
```
means = np.mean(data, axis=0)
stds = np.std(data, axis=0)
data_norm = (data - means) / stds
```

In [26]:
```
# This step for separate data
def train_test(X,Y,size):
    rng = default_rng()
    SizeTrain=int((size*X.shape[0]))
    random_np=rng.choice(X.shape[0], size=SizeTrain, replace=False)
    X_test=X[random_np,:]
    y_test=Y[random_np]
    Y_train=np.delete(Y, random_np,axis=0)
    X_train=np.delete(X, random_np,axis=0)
    return X_train,X_test,Y_train,y_test
# Extract y from the normalized dataset

y_index = 2
y = np.array([data_norm[:,1]]).T

# Extract X from normalized dataset

X = data_norm[:,0:2]
print(np.shape(X))
X_train,X_test,Y_train,y_test = train_test(X,y,0.3)
# Insert column of 1's for intercept term

X_train = np.insert(X_train, 0, 1, axis=1)
X_test = np.insert(X_test, 0, 1, axis=1)
```

```
(117, 2)
```

check number of datasets x

In [35]:
```
m = X_train.shape[0]
n = X_train.shape[1]
print(m, n)
```

```
82 3
```

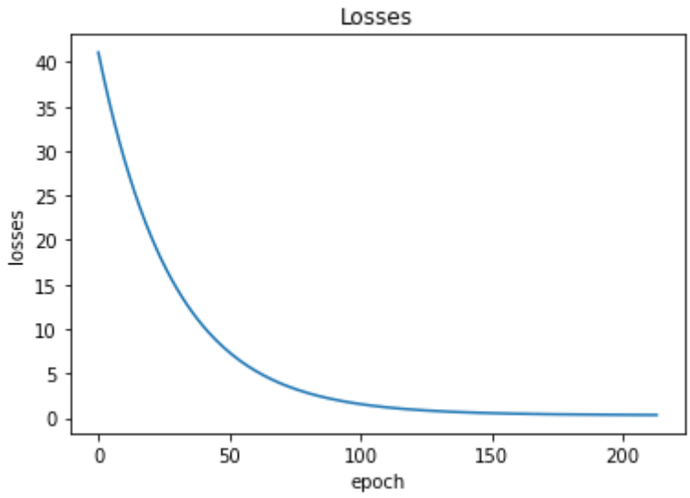
This step i use gradient descent for optimization by alpha = alpha = 0.0001 iterations = 1000000 and calcurate will stop when theta move less than 0.001

In [43]:
```
#m = y.shape[0]


def cost(theta, X, y):

    dy=(np.dot(X,theta)-y)

    J = 0.5*np.dot(dy.T,dy)

    return J
def h_theta(X, theta): # theta dot X
    theta=theta.reshape(3,)

    print(np.shape(X))
    print(np.shape(theta))
    return np.dot(X,theta)
def gradient_descent(X, y, theta_initial, alpha, num_iters):
    J_per_iter = np.zeros(num_iters)
    gradient_per_iter = np.zeros((num_iters,len(theta_initial)))
    # initialize theta
    theta = theta_initial
    loss_old = 10000
    tol = 0.001
    loss=[]
    start = time()
    for iter in np.arange(num_iters):
        # YOUR CODE HERE
        grad=gradient(X, y, theta)
        theta = theta - alpha * grad
        #raise NotImplementedError('ValueError')
        J_per_iter[iter] = cost(theta, X, y)
        loss_current = J_per_iter[iter]
        different=np.abs(loss_old-loss_current)
        if different == tol or different < tol:
            break
        loss_old=loss_current
        gradient_per_iter[iter] = grad.T
        loss.append(loss_old)
    #print(loss)
    #loss.reshape(loss.shape(0))
    time_taken = time() - start
    print('Times : {} s'.format(time_taken))
    #yhat = np.argmax(h_theta(X_test, theta), axis=1)

    cost_plot(loss)
    return theta, J_per_iter, gradient_per_iter

def gradient(X, y, theta):
    # YOUR CODE HERE
    X_theta= np.dot(X,theta)
    grad = np.dot(X.T,(X_theta-y))

    #raise NotImplementedError()
    return grad
def cost_plot(loss):
    #def cost_plot(iterations, costs):
    plt.plot(np.arange(len(loss)),loss,label= ' Train losses')
    plt.title('Losses')
    plt.xlabel('epoch')
    plt.ylabel('losses')
def h(X,theta):
    #theta=theta.reshape(3,)
    y_pre=np.dot(X,theta)
    return y_pre
def goodness_of_fit(y, y_predicted):
    y_predicted=y_predicted.reshape(np.shape(y))
    r_square = 1-((np.sum((y-y_predicted)**2))/(np.sum((y-np.mean(y_pred
icted))**2)))
    #print(np.shape(y))
    return r_square
```

In [44]:
```
theta_initial = np.zeros((X_train.shape[1],1))
alpha = 0.0001
iterations = 600
```

In [45]:
```
theta, costs, grad = gradient_descent(X_train,Y_train, theta_initial, al
pha, iterations)
# system will get a train losses chart  and time of calculation
```

```
Times : 0.015558481216430664 s
```



In [46]:
```
# Goodness of fit
y_predicted = h(X_test,theta)
# YOUR CODE HERE
r_square = goodness_of_fit(y_test,y_predicted)
#raise NotImplementedError()
print('Theta values ', theta)
print('r_square : {}'.format(r_square))
```

```
Theta values  [[0.01296217]
 [0.47441071]
 [0.48764956]]
r_square : 0.9826781891092137
```

This step for normal equation with

In [47]:
```
def h(X,theta):
    theta=theta.reshape(3,1)
    y_pre=np.dot(X,theta)
    return y_pre
def goodness_of_fit(y, y_predicted):
    y_predicted=y_predicted.reshape(np.shape(y))
    r_square = 1-((np.sum((y-y_predicted)**2))/(np.sum((y-np.mean(y_pred
icted))**2)))
    #print(np.shape(y))
    return r_square
from numpy.linalg import inv
def normal_equation(X, y):
    num_iters=30000000
    alpha = 0.000001
    theta_initial=np.array([0, 1, X.shape[0]])
    J_per_iter = np.zeros(num_iters)
    gradient_per_iter = np.zeros((num_iters,len(theta_initial)))

    # initialize theta
    #theta = theta_initial
    X_dot= np.dot(X.T,X)
    X_dot_inv=np.linalg.inv(X_dot)
    X_dot_y=X.T@y
    theta =X_dot_inv @X_dot_y
    #for iter in np.arange(num_iters):
        # YOUR CODE HERE
        #grad=gradient(X, y, theta)

        #costs=cost(theta, X, y)
        #J_per_iter[iter] = cost(theta, X, y)
        #gradient_per_iter[iter] = grad.T
    #return (theta, J_per_iter, gradient_per_iter)
    return theta.T
```

In [48]:
```
theta_norm = normal_equation(X_train,Y_train)
print("theta from normal equation:", theta_norm.T)
y_norm_predicted =  h(X_test, theta_norm)
r_norm_square = goodness_of_fit(y_test, y_norm_predicted)
print("r_square:", r_norm_square)
```

```
theta from normal equation: [[-1.30104261e-16]
 [-7.10542736e-15]
 [ 1.00000000e+00]]
r_square: 1.0
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: