Before you turn this problem in, make sure everything runs as expected. First, **restart the kernel** (in the menubar, select Kernel$\rightarrow$Restart) and then **run all cells** (in the menubar, select Cell$\rightarrow$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 = "Aung Zar Lin"
ID = "121956"
```

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:
\begin{equation} y\sim\mathcal{N}(\theta_0+\theta_1x,\sigma^2) \end{equation}

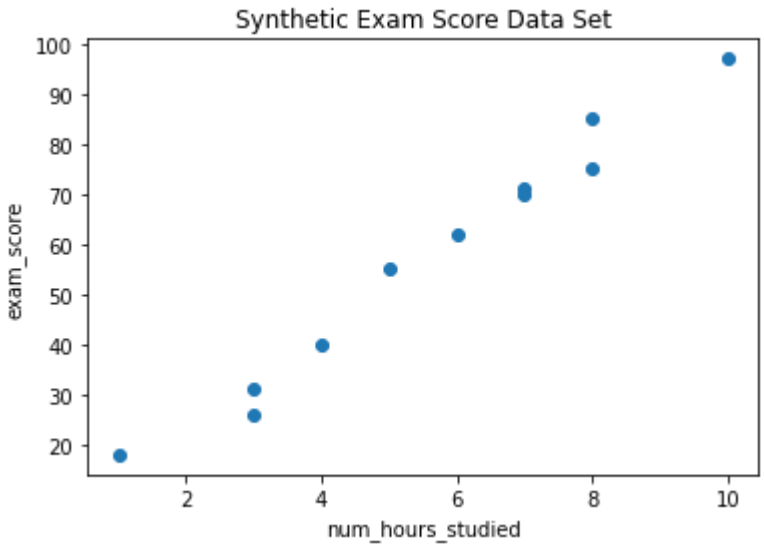
Our model or hypothesis, then, will be a function predicting y based on x: \begin{equation} h_\theta(x)=\theta_0+\theta_1x \end{equation}

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 $\mathtt{X}$, 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
\begin{equation} \mathtt{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} \end{equation}

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

Now we can conveniently write the hypothesis as
$$ h_\mathbf{\theta}(\mathbf{x}) = \mathbf{\theta}^\top \mathbf{x} . $$

Exercise 1 (2 points)

Write a Python code function to evaluate a hypothesis $\mathbf{\theta}$ for an entire design matrix:

Hint: Use numpy function of `dot`

In [5]:

```
# Evaluate hypothesis over a design matrix

def h(X,theta):
    # YOUR CODE HERE
    y_predicted = np.dot(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 $\mathbf{\theta}$? 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:
$$\begin{align} \ J(\mathbf{\theta}) & = \frac{1}{2} \sum_{i=1}^{m}\left(h_\mathbf{\theta}\left(\mathbf{x}^{(i)}\right) - y^{(i)}\right)^2 \\ \ & = \frac{1}{2} \left(\mathtt{X} \mathbf{\theta} - \mathbf{y} \right)^\top \left(\mathtt{X} \mathbf{\theta} - \mathbf{y} \right) \end{align}$$

Here we've used $\mathtt{X}$ 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 $dy = \hat{y} - y = \mathtt{X}\theta - y$
2. Calcuate $cost = \frac{1}{2}{dy}^T{dy}$

In [7]:

```
m = y.shape[0]

def cost(theta, X, y):
    # YOUR CODE HERE
    dy = h(X, theta) - y
    J = np.dot(dy.T, dy) / 2
    #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 $J(\mathbf{\theta})$ above with respect to the parameters $\mathbf{\theta}$.

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 $\mathbf{\theta}$.

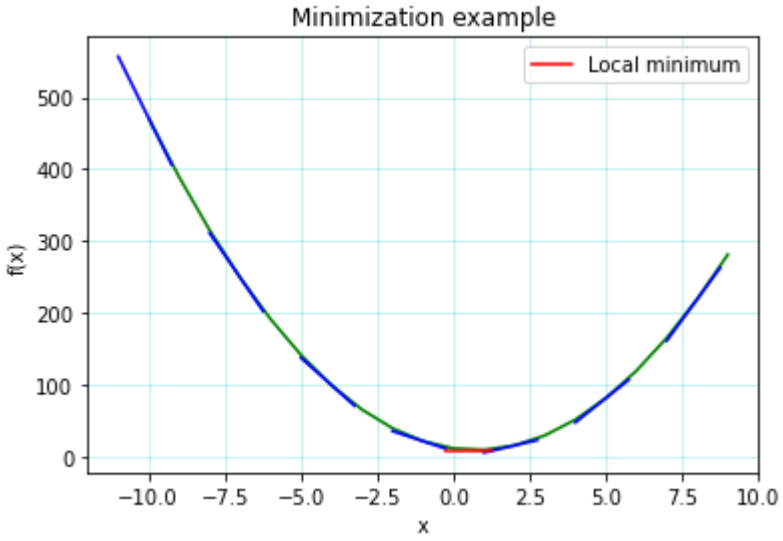
To find this minimum, we will find the point at which the gradient is equal to the zero vector.

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

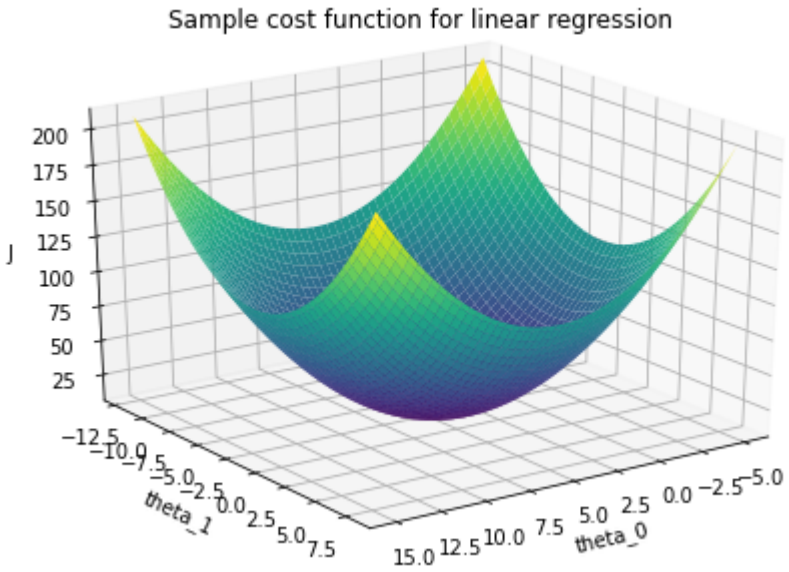
Based on the previous example, we can see that to minimize our cost function, we just need to take the gradient with respect to $\mathbf{\theta}$ and determine where that gradient is equal to $\mathbf{0}$.

We have$$ J(\mathbf{\theta}) = \frac{1}{2} \sum_{i=1}^{m} \left(h_\mathbf{\theta}(\mathbf{x}^{(i)}) - y^{(i)}\right)^2 .$$This is a convex function of two variables (θ_0 and θ_1), so it has a single minimum where the gradient $\nabla_J(\mathbf{\theta})$ 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 $\mathbf{\theta}$, you get
$$ \nabla_J(\mathbf{\theta}) = \mathtt{X}^\top (\mathtt{X}\mathbf{\theta}-\mathbf{y}).$$

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
    grad = np.dot(X.T, h(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()
        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 84.44605981 84.3
8279953
 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]]

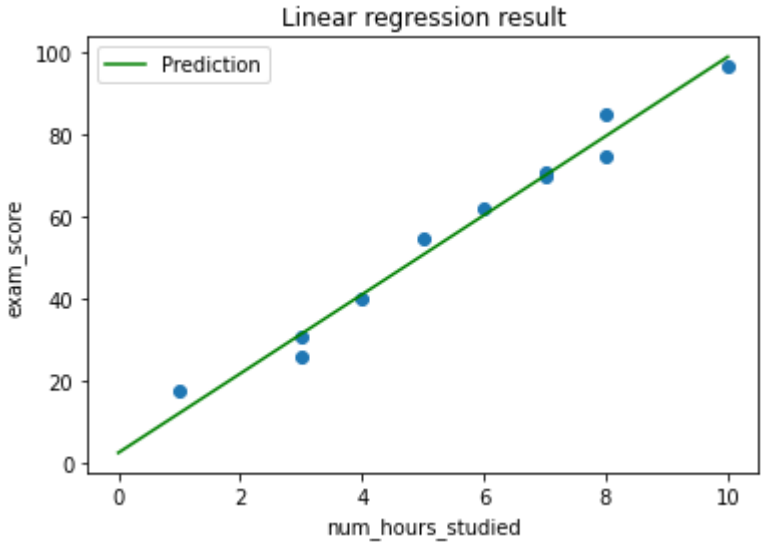
In [15]:

```
# 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, iterations)
print('Optimal parameters: theta_0 %f theta_1 %f' % (theta[0], theta[1]))
```

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

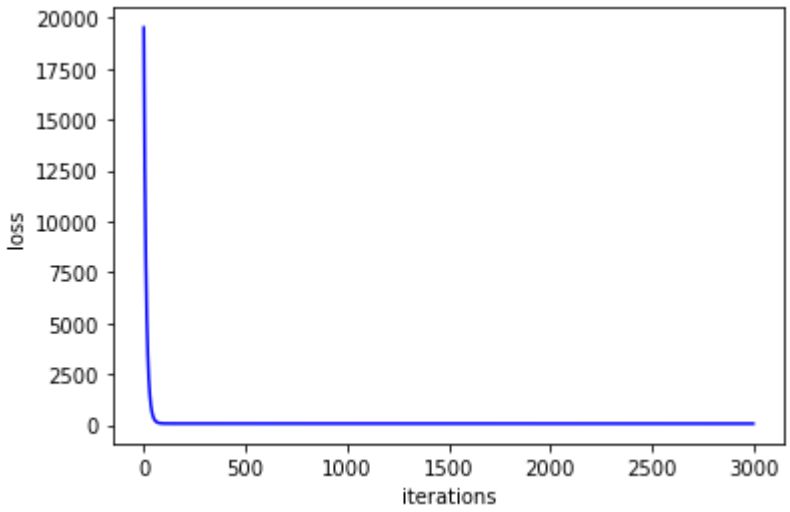
In [16]:

```
# 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)

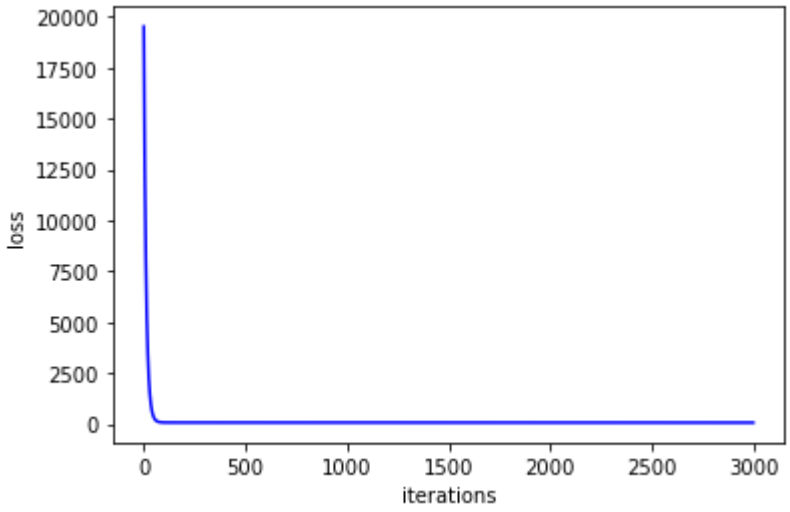
Instead of repeating the code to plot the loss graph, we would like to encapsulate the code in a function. Complete the loss plotting function below:

In [18]:

```
def cost_plot(iterations, costs):
    # YOUR CODE HERE
    plt.plot(np.arange(iterations), costs, 'b-')
    plt.xlabel('iterations')
    plt.ylabel('loss')
    plt.show()
    # raise NotImplementedError()
```

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

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):
    # YOUR CODE HERE
    r_square = 1 - ((np.square(y - y_predicted)).sum()/(np.square(y - y.mean())).sum())
    # raise NotImplementedError()
    return r_square
```

In [21]:

```
y_predicted =  h(X, theta)
r_square = goodness_of_fit(y, y_predicted)
print(r_square)
```

```
0.9786239731773175
```

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 file.

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:

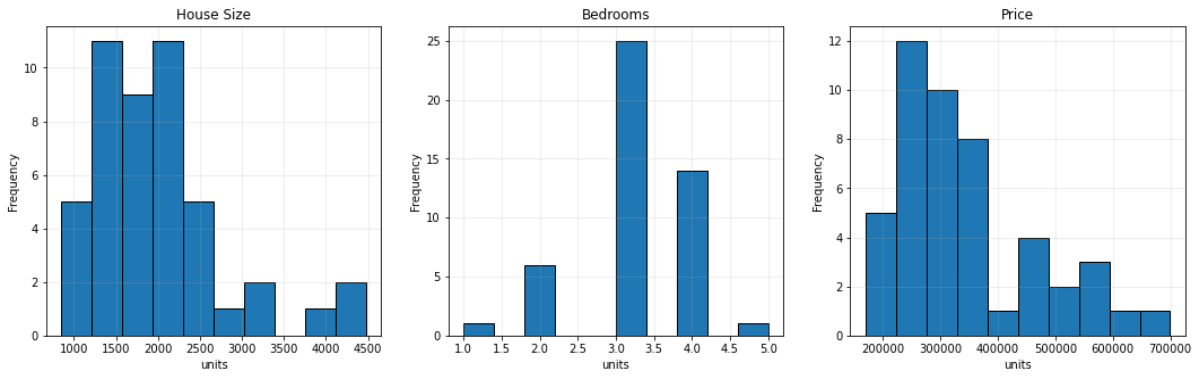
In [23]:

```
# 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]:

```
theta_initial = np.zeros((X.shape[1],1))
alpha = 0.01
iterations = 1000
theta, costs, grad = gradient_descent(X, y, theta_initial, alpha, iterations)
# YOUR CODE HERE
# raise NotImplementedError()
```

In [29]:

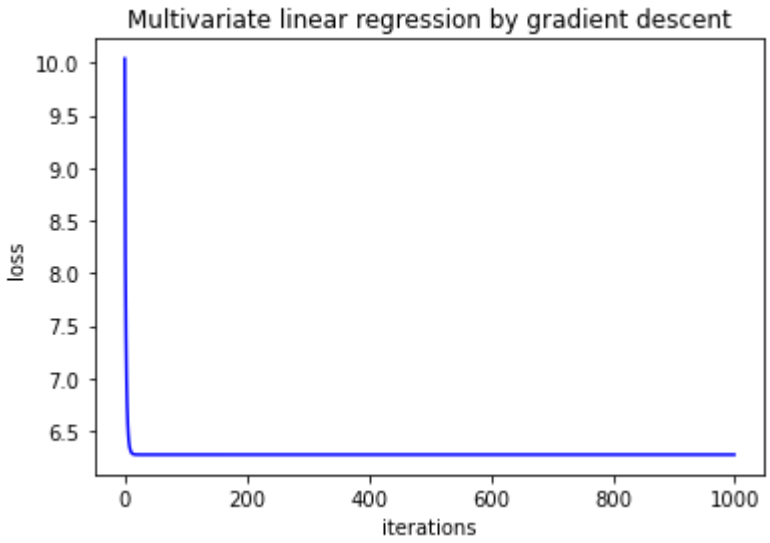
```
print('Theta values ', theta)
```

```
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 [30]:

```
# 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
$$\hat{y}^{\text{norm}} = \theta^\text{norm} \textbf{x}^\text{norm}$$

Excercise 8 (3 points)

Modify the code to compute goodness of fit

In [31]:

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

In [32]:

```
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$$y*{\text{norm}} = \sigma_y y + \mu_y$$

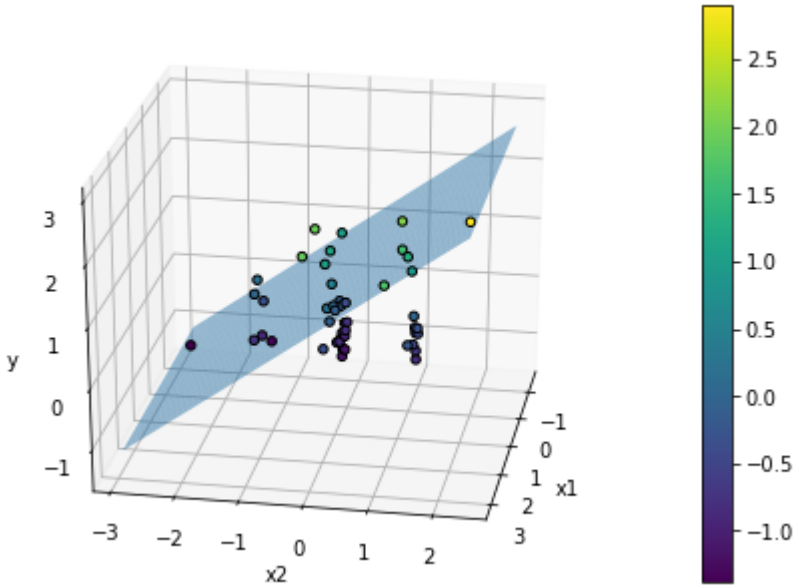
In [33]:

```
# 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 [34]:

```
# 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 [35]:

```
# just remove all parameters
%reset
```

In [36]:

```
import matplotlib.pyplot as plt
import numpy as np
```

Exercise 2.1 (5 points)

Download raw_data and setup data

In [37]:

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

In [38]:

```
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 [39]:

```
# Normalized data
def normalized_data(data):
    # YOUR CODE HERE
    return (data - np.mean(data, axis=0)) / np.std(data, axis=0)
    #raise NotImplementedError()
```

In [40]:

```
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 [41]:

```
# Extract y from data
# YOUR CODE HERE
y_label = 'Price';
y_index = np.where(headers == y_label)[0][0]
y = data_norm[:, y_index]
#raise NotImplementedError()
```

In [42]:

```
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 [43]:

```
# Extract X from data
# YOUR CODE HERE
X = data_norm[:, 0:y_index]
X = np.insert(X, 0, 1, axis=1)
#raise NotImplementedError()
```

In [44]:

```
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 [45]:

```
# create h function
def h(X,theta):
    # YOUR CODE HERE
    y_predicted = np.dot(X, theta)
    #raise NotImplementedError()
    return y_predicted
```

In [46]:

```
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 [47]:

```
def cost(theta, X, y):
    # YOUR CODE HERE
    dy = h(X, theta) - y
    J = np.dot(dy.T, dy) / 2
    #raise NotImplementedError()
    return J
```

In [48]:

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

```
5477.13862837469
```

Expected result: 5477.138628374691

In [49]:

```
# Gradient of cost function
def gradient(X, y, theta):
    # YOUR CODE HERE
    grad = np.dot(X.T, h(X, theta) - y)
    # raise NotImplementedError()
    return grad
```

In [50]:

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

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

Expected result: [47. 599.00016917 659.76139633]

In [51]:

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

In [52]:

```
(theta, J_per_iter, gradient_per_iter) = gradient_descent(X, y, np.array([0, 1, 10]), 0.00
1, 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 1305.658
34104
 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 [53]:

```
# YOUR CODE HERE
alpha = 0.003
iterations = 30000
theta_initial = np.zeros((X.shape[1],))
theta, costs, grad = gradient_descent(X, y, theta_initial, alpha, iterations)
# raise NotImplementedError()
```

In [54]:

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

```
theta: [-7.69384556e-17  8.84765988e-01 -5.31788197e-02]
cost_per_iter: [6.27579208 6.27579208 6.27579208 6.27579208 6.27579208]
gradient_per_iter [[-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]
 [-1.38777878e-16 -1.78468351e-14  1.15185639e-15]
 [ 3.05311332e-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 [55]:

```
def goodness_of_fit(y, y_predicted):
    # YOUR CODE HERE
    r_square = 1 - ((np.square(y - y_predicted.T)).sum()/(np.square(y - y.mean())).sum())
    # raise NotImplementedError()
    return r_square
```

In [56]:

```
y_predicted =  h(X, theta)
r_square = goodness_of_fit(y, y_predicted)
print(r_square)
```

```
0.7329450180289143
```

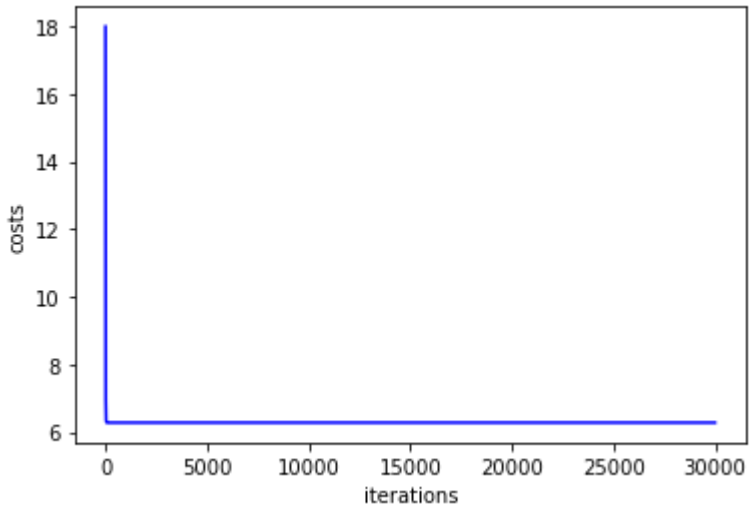
Expected result: 0.7329450180289143

Excercise 2.7 (2 point)

Plot graph of cost results

In [57]:

```
def cost_plot(iterations, costs):
    # YOUR CODE HERE
    plt.plot(np.arange(iterations), costs, 'b-')
    plt.xlabel('iterations')
    plt.ylabel('costs')
    plt.show()
    #raise NotImplementedError()

cost_plot(iterations, costs)
```



Exercise 2.8 (8 points)

Write a function implementing the normal equations for linear equation:

In [58]:

```
# Function to use the normal equations to find the optimal
# parameters for a linear regression model

def normal_equation(X, y):
    # YOUR CODE HERE
    theta = np.linalg.inv(X.T.dot(X)).dot(X.T.dot(y))
    # aise NotImplementedError()
    return theta
```

In [59]:

```
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-0
2]]
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?

YOUR ANSWER HERE

In this exercise, the Medical Cost Personal Datasets was used which has 1338 sample size, 7 features (6 input features and 1 output variable). After inputing the data, the first step is to do Data Analysis. The data type of each columns was checked and also whethere they have null values or not, and found out there is no null values in this dataset. Then, columns with object dtypes were converted into int64 or float64. Next step is data visualization to see the distribution of the data. In this step, the data was normalized and split into training(70%) and test(30%) sets. The training set was fitted into the model by using both gradient descent and normal equation. In gradient descent, the parameter learning rate was set to 0.0001 and run over iterations 100, 1000, 10000. The training loss (cost) was recorded in each iteration and plot in the graph to see the loss over iterations. From the graph, we can see that after 40 iterations, the difference between the previous cost and the current cost was nearly zero. Setting iterations to 50 is enough for this model and it will reduce the computation time. Then y_predicted was computed with test set and updated theta value from gradient descent and normal equation. Then r_squared value was computed. r_square value from gradient descent is 0.7417255854352769 and from normal equation is 0.7415943033969101, there is no significantly difference between them.

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

In [60]:

```
# YOUR CODE HERE
%reset
#raise NotImplementedError()
```

In [61]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Data Analysis

In [62]:

```
data = pd.read_csv('insurance.csv')
data.head(5)
```

Out[62]:

	age	sex	bmi	children	smoker	region	charges
0	19	female	27.900	0	yes	southwest	16884.92400
1	18	male	33.770	1	no	southeast	1725.55230
2	28	male	33.000	3	no	southeast	4449.46200
3	33	male	22.705	0	no	northwest	21984.47061
4	32	male	28.880	0	no	northwest	3866.85520

In [63]:

```
print(data.shape)
```

```
(1338, 7)
```

In [64]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   age       1338 non-null   int64  
 1   sex       1338 non-null   object 
 2   bmi       1338 non-null   float64
 3   children  1338 non-null   int64  
 4   smoker    1338 non-null   object 
 5   region    1338 non-null   object 
 6   charges   1338 non-null   float64
dtypes: float64(2), int64(2), object(3)
memory usage: 73.3+ KB
```

In [65]:

```
data.describe()
```

Out[65]:

	age	bmi	children	charges
count	1338.000000	1338.000000	1338.000000	1338.000000
mean	39.207025	30.663397	1.094918	13270.422265
std	14.049960	6.098187	1.205493	12110.011237
min	18.000000	15.960000	0.000000	1121.873900
25%	27.000000	26.296250	0.000000	4740.287150
50%	39.000000	30.400000	1.000000	9382.033000
75%	51.000000	34.693750	2.000000	16639.912515
max	64.000000	53.130000	5.000000	63770.428010

In [66]:

```
data.isnull().sum()
```

Out[66]:

```
age         0
sex         0
bmi         0
children    0
smoker      0
region      0
charges     0
dtype: int64
```

Since there is no null value, we don't need to fill any missing data.

Preparing Data for the model

First, we will convert all values to type int or float so that the model can process them. (some columns have object as dtype)

Categorial Features

- sex
- smoker
- region

In [67]:

```
print(data['sex'].unique())
print(data['smoker'].unique())
print(data['region'].unique())
```

```
['female' 'male']
['yes' 'no']
['southwest' 'southeast' 'northwest' 'northeast']
```

Next, we will convert the 'sex' column to 0.0 for Male and 1.0 for Female.

In [68]:

```
data['sex'].replace('male', 0.0, inplace = True)
data['sex'].replace('female', 1.0, inplace = True)
```

Next, we will convert the 'smoker' column to 0.0 for No and 1.0 for Yes

In [69]:

```
data['smoker'].replace('no', 0.0, inplace = True)
data['smoker'].replace('yes', 1.0, inplace = True)
```

Next, we will convert the 'region' column to 1.0 to northeast, 2.0 to southeast, 3.0 southwest and 4.0 to northwest.

In [70]:

```
data['region'].replace('northeast', 1.0, inplace = True)
data['region'].replace('southeast', 2.0, inplace = True)
data['region'].replace('southwest', 3.0, inplace = True)
data['region'].replace('northwest', 4.0, inplace = True)
```

In [71]:

```
data.head(5)
```

Out[71]:

	age	sex	bmi	children	smoker	region	charges
0	19	1.0	27.900	0	1.0	3.0	16884.92400
1	18	0.0	33.770	1	0.0	2.0	1725.55230
2	28	0.0	33.000	3	0.0	2.0	4449.46200
3	33	0.0	22.705	0	0.0	4.0	21984.47061
4	32	0.0	28.880	0	0.0	4.0	3866.85520

In [72]:

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1338 entries, 0 to 1337
Data columns (total 7 columns):
 #   Column    Non-Null Count  Dtype  
---  ------    --------------  -----  
 0   age       1338 non-null   int64  
 1   sex       1338 non-null   float64
 2   bmi       1338 non-null   float64
 3   children  1338 non-null   int64  
 4   smoker    1338 non-null   float64
 5   region    1338 non-null   float64
 6   charges   1338 non-null   float64
dtypes: float64(5), int64(2)
memory usage: 73.3 KB
```

All data is either float or int now!

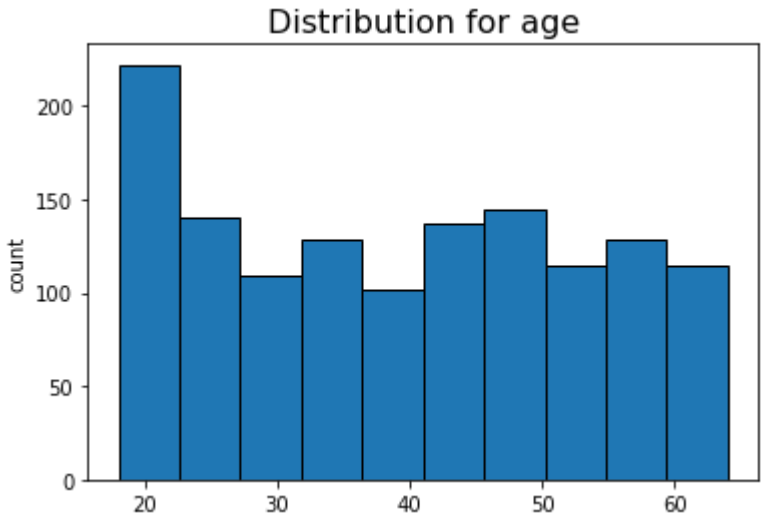
Data Visualization

In [73]:

```
def histplot(data, name):
    plt.hist(data[name], edgecolor='black')
    plt.title(f"Distribution for {name}", size=16)
    plt.ylabel('count')
    plt.show()
```

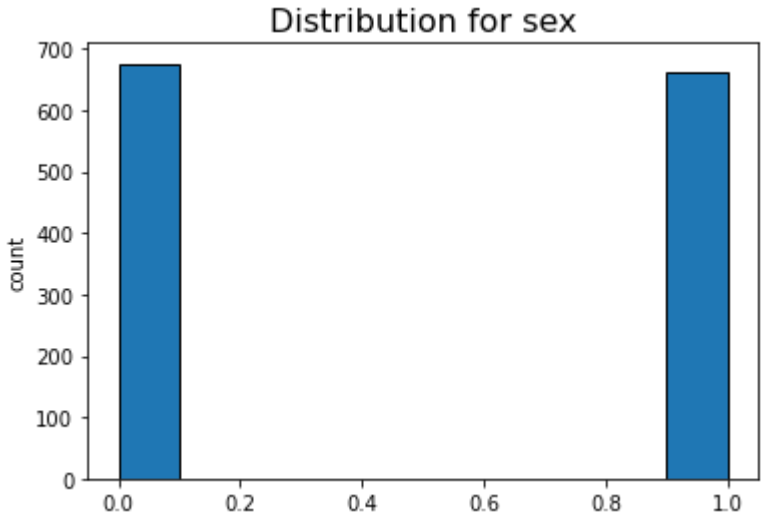
In [74]:

```
histplot(data, 'age')
```



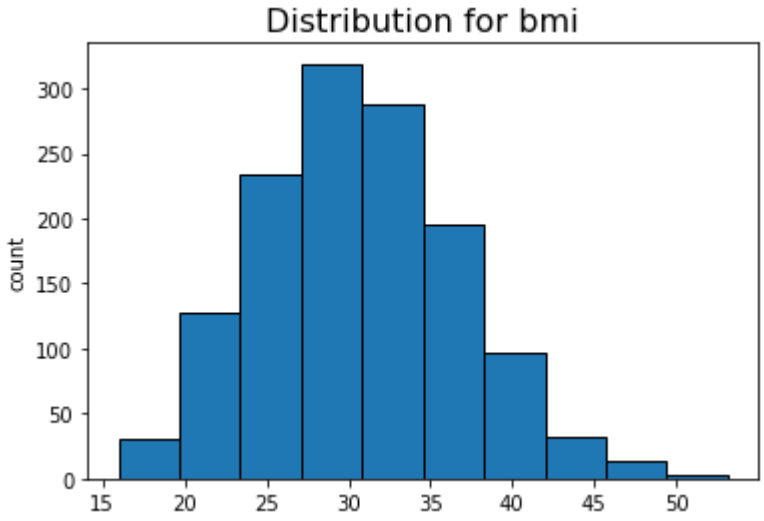
In [75]:

```
histplot(data, 'sex')
```



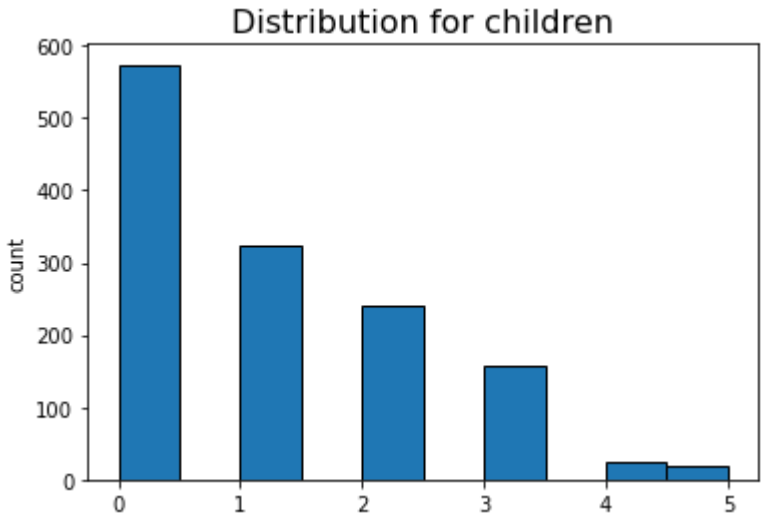
In [76]:

```
histplot(data, 'bmi')
```



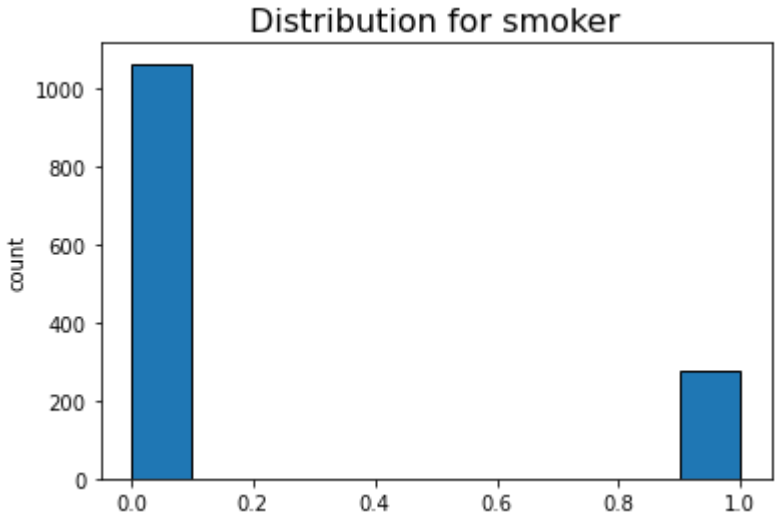
In [77]:

```
histplot(data, 'children')
```



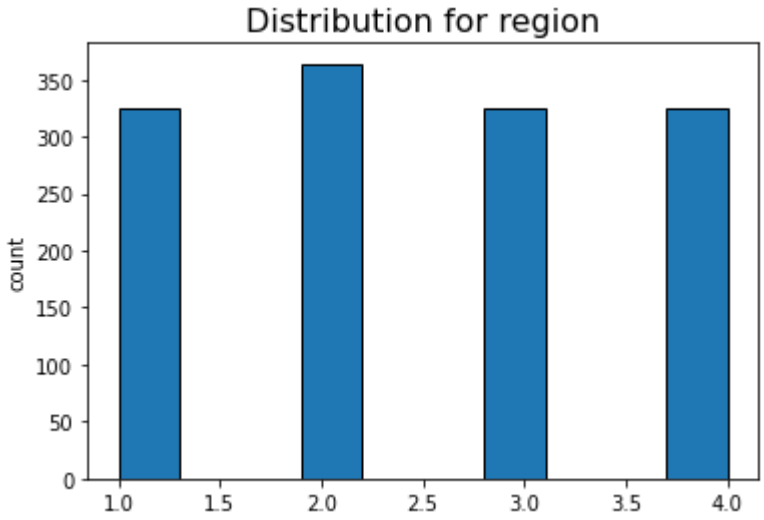
In [78]:

```
histplot(data, 'smoker')
```



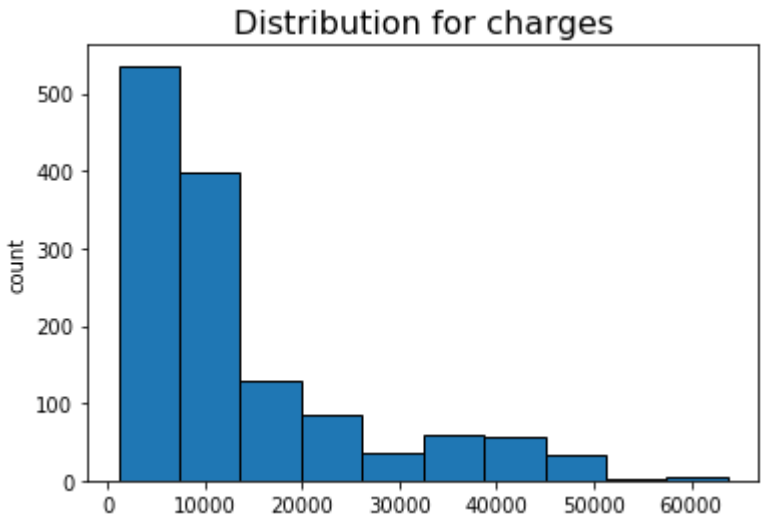
In [79]:

```
histplot(data, 'region')
```



In [80]:

```
histplot(data, 'charges')
```



Normalizing the data

In [81]:

```
means = np.mean(data, axis=0)
stds = np.std(data, axis=0)
data_norm = (data - means) / stds
```

In [82]:

```
data_norm.head(5)
```

Out[82]:

	age	sex	bmi	children	smoker	region	charges
0	-1.438764	1.010519	-0.453320	-0.908614	1.970587	0.464873	0.298584
1	-1.509965	-0.989591	0.509621	-0.078767	-0.507463	-0.440513	-0.953689
2	-0.797954	-0.989591	0.383307	1.580926	-0.507463	-0.440513	-0.728675
3	-0.441948	-0.989591	-1.305531	-0.908614	-0.507463	1.370259	0.719843
4	-0.513149	-0.989591	-0.292556	-0.908614	-0.507463	1.370259	-0.776802

In [83]:

```
y = data_norm['charges']

X = data_norm.drop(columns='charges', axis=1)
```

In [84]:

```
X = np.array(X)
y = np.array([y]).T

print(X.shape)
print(y.shape)
```

```
(1338, 6)
(1338, 1)
```

In [85]:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3)
print("X_train shape", X_train.shape)
print("X_test shape", X_test.shape)
print("y_train shape", y_train.shape)
print("y_test shape", y_test.shape)
```

```
X_train shape (936, 6)
X_test shape (402, 6)
y_train shape (936, 1)
y_test shape (402, 1)
```

In [86]:

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

print("X_train shape", X_train.shape)
print("X_test shape", X_test.shape)
```

```
X_train shape (936, 7)
X_test shape (402, 7)
```

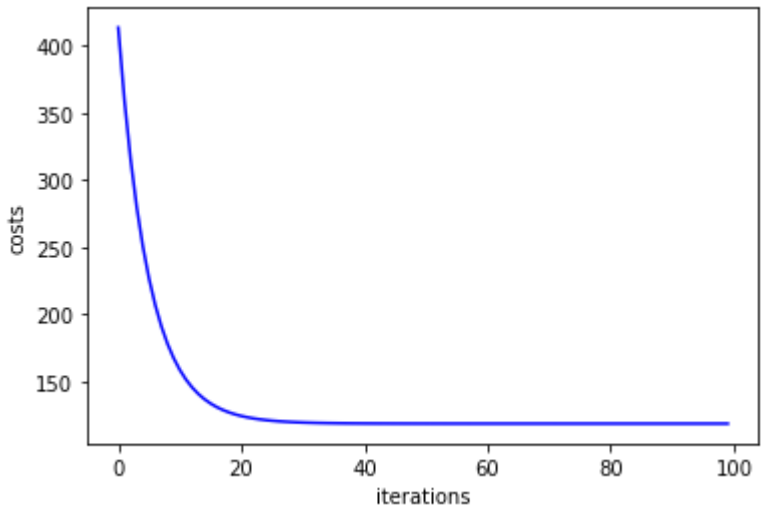
In [87]:

```
def h_theta(X, theta):
    y_predicted = X.dot(theta)
    return y_predicted

def cost_function(X, y, theta):
    dy = h_theta(X, theta) - y
    J = dy.T.dot(dy) / 2
    return J

def gradient(X, y, theta):
    grad = X.T.dot(h_theta(X, theta) - y)
    return grad

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)))
    theta = theta_initial
    for ix in np.arange(num_iters):
        grad = gradient(X, y, theta)
        theta = theta - alpha * grad
        J_per_iter[ix] = cost_function(X, y, theta)
        gradient_per_iter[ix] = grad.T
    return (theta, J_per_iter, gradient_per_iter)

def goodness_of_fit(y, y_pred):
    return 1 - (np.square(y - y_pred).sum()) / (np.square(y - y.mean()).sum())

def cost_plot(iterations, costs):
    plt.plot(np.arange(iterations), costs, 'b-')
    plt.xlabel('iterations')
    plt.ylabel('costs')
    plt.show()

def normal_equation(X, y):
    theta = np.linalg.inv(X.T.dot(X)).dot(X.T.dot(y))
    return theta
```

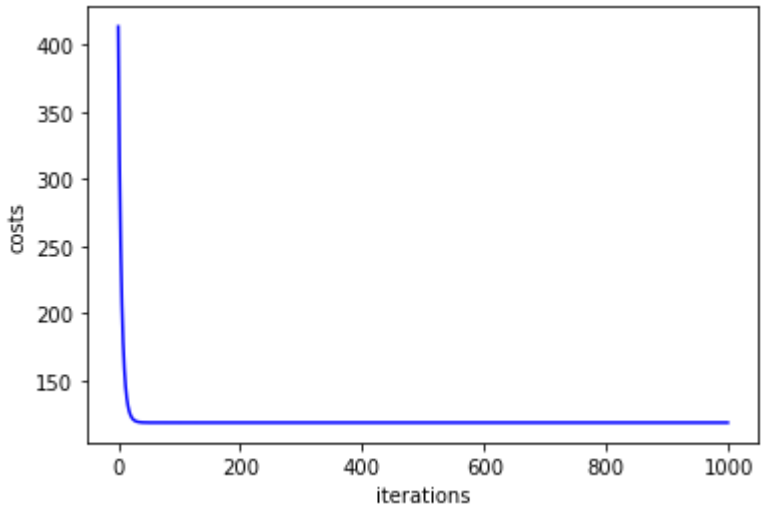
In [88]:

```
theta_initial = np.zeros((X_train.shape[1],1))

alpha = [0.0001, 0.0001, 0.0001]
iterations = [100, 1000, 10000]
for i in range(0, len(alpha)):
    theta, costs, grad = gradient_descent(X_train, y_train,theta_initial,alpha[i],iteratio
ns[i])
    print('Alpha =', alpha[i])
    print('Iterations =', iterations[i])
    print('Theta = ', theta)
    cost_plot(iterations[i], costs)
```

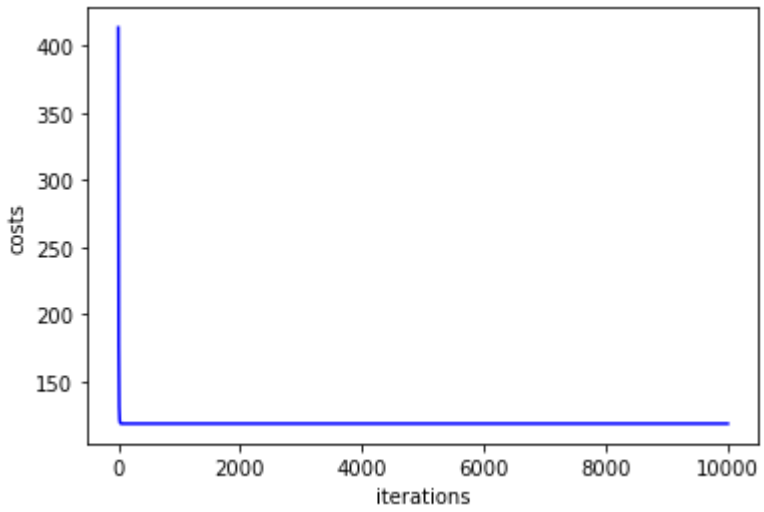
```
Alpha = 0.0001
Iterations = 100
Theta =  [[ 0.00905241]
 [ 0.2988685 ]
 [-0.0048088 ]
 [ 0.15807644]
 [ 0.04546464]
 [ 0.80074026]
 [-0.00829901]]
```




```
Alpha = 0.0001
Iterations = 1000
Theta =  [[ 0.00905871]
 [ 0.29888224]
 [-0.00476772]
 [ 0.15808705]
 [ 0.04545394]
 [ 0.80080414]
 [-0.00826391]]
```



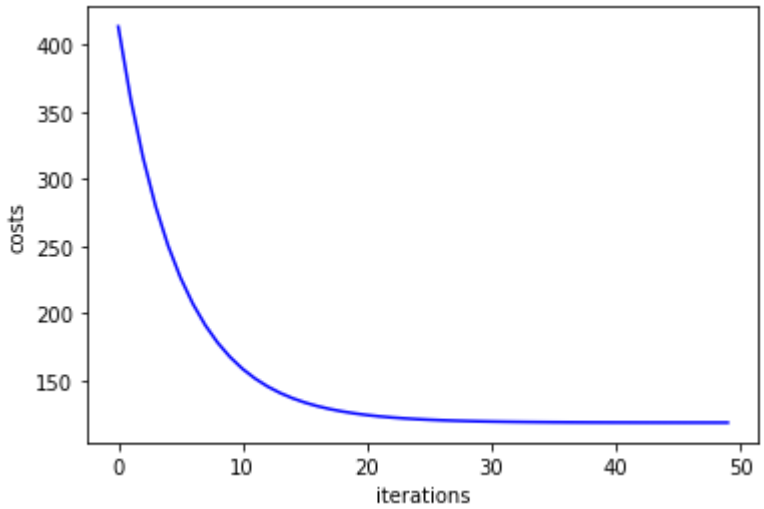
```
Alpha = 0.0001
Iterations = 10000
Theta =  [[ 0.00905871]
 [ 0.29888224]
 [-0.00476772]
 [ 0.15808705]
 [ 0.04545394]
 [ 0.80080414]
 [-0.00826391]]
```



In [89]:

```
alpha = 0.0001
iterations=50
theta, costs, grad = gradient_descent(X_train, y_train,theta_initial,alpha,iterations)
print(theta)
cost_plot(iterations, costs)
```

```
[[ 0.00867857]
 [ 0.29685831]
 [-0.00714016]
 [ 0.15752406]
 [ 0.04595349]
 [ 0.79435085]
 [-0.01025013]]
```



In [90]:

```
y_pred = h_theta(X_test, theta)
r_square = goodness_of_fit(y_test, y_pred)
print(r_square)
```

```
0.7417255854352769
```

In [91]:

```
optimal_theta = normal_equation(X_train, y_train)
print(optimal_theta)
```

```
[[ 0.00905871]
 [ 0.29888224]
 [-0.00476772]
 [ 0.15808705]
 [ 0.04545394]
 [ 0.80080414]
 [-0.00826391]]
```

In [92]:

```
y_predicted = h_theta(X_test, optimal_theta)
r_square = goodness_of_fit(y_test, y_predicted)
print(r_square)
```

```
0.7415943033969101
```

In [ ]:

```

```