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 [1]:
```
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:

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

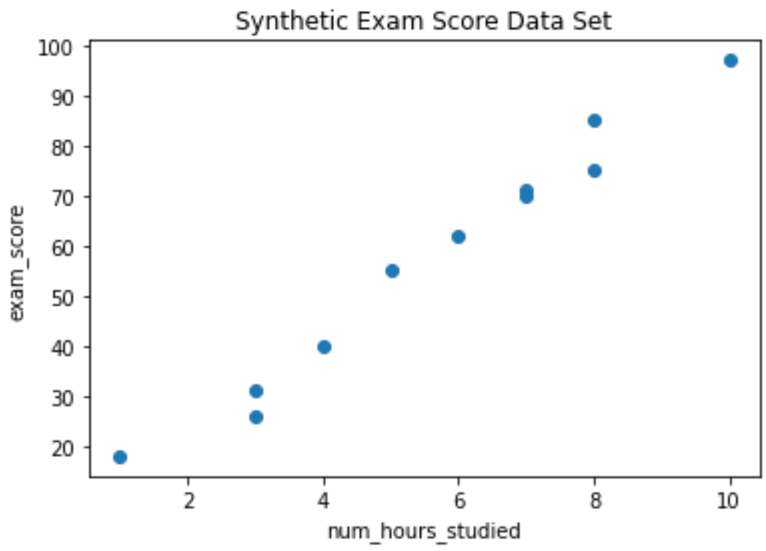
Our model or hypothesis, then, will be a function predicting y based on x:

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

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 [2]:
```
# 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, 9
7])

# 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

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

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

In [3]:
```
# Add dummy variable for intercept term to design matrix.
# Understand the numpy insert function by reading https://numpy.o
rg/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 point)

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

Hint: Use numpy function of `dot`

In [4]:
```
# Evaluate hypothesis over a design matrix

def h(X,theta):
    ### BEGIN SOLUTION
    y_predicted = X.dot(theta)
    ### END SOLUTION
    return y_predicted
```

In [5]:
```
print(h(X, np.array([0, 10])))
### BEGIN HIDDEN TESTS
res = h(X, np.array([0, 10]))
assert res.shape == (11,), "Data size in result is incorrect"
assert res[4] == 50, "Data result is incorrect"
assert np.array_equal(h(X, np.array([3, 10])), [ 13, 33, 33, 43,
53, 63, 73, 73, 83, 83, 103]), "Function h is incorrect"
### END HIDDEN TESTS
```

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

Expect 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:

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

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 point)

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 [6]:
```
m = y.shape[0]

def cost(theta, X, y):
    ### BEGIN SOLUTION
    y_predicted = h(X, theta) - y
    J = y_predicted.T.dot(y_predicted)/2
    ### END SOLUTION
    return J
```

In [7]:
```
print(cost(np.array([0, 10]), X, y))
### BEGIN HIDDEN TESTS
assert cost(np.array([0, 10]), X, y) == 85.0, "Data result is inc
orrect"
assert cost(np.array([3, 10]), X, y) == 104.5, "Function cost is
incorrect"
### END HIDDEN TESTS
```

```
85.0
```

Expect 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.

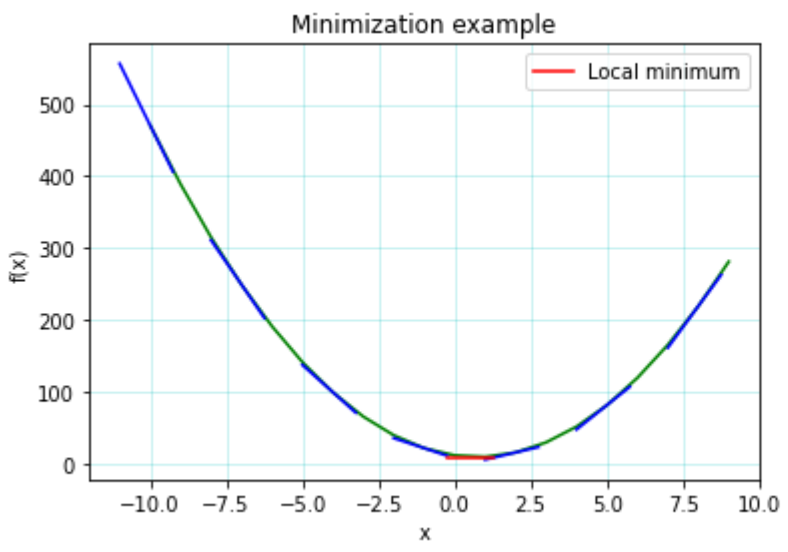
J is convex (see `[Wikipedia](https://en.wikipedia.org/wiki/Convex_function)` for an explanation) so it has just one minimum for some specific value of θ.

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 [8]:
```
# 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 $\mathbf{0}$.

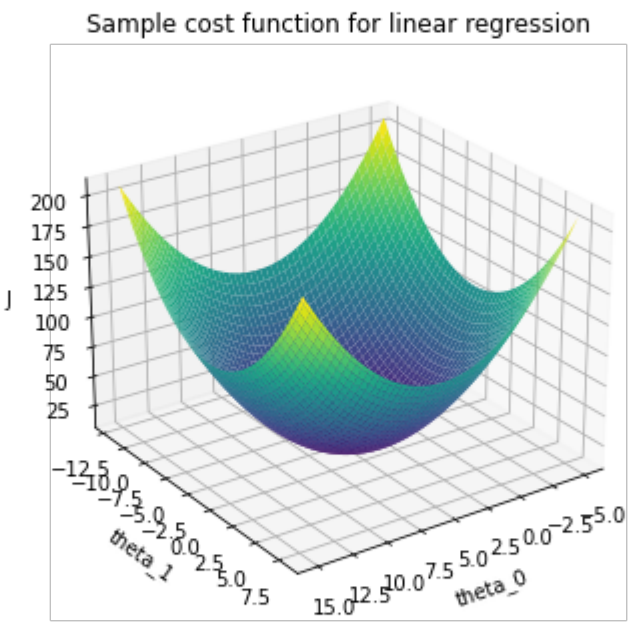
We have

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

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 [9]:
```
# 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

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

Exercise 3 (2 point)

Write **gradient function** in python code from equation above:

In [10]:
```
# Gradient of cost function
def gradient(X, y, theta):
    ### BEGIN SOLUTION
    grad = X.T.dot(h(X, theta) - y)
    ### END SOLUTION
    return grad
```

In [11]:
```
print(gradient(X, y, np.array([0, 10])))
### BEGIN HIDDEN TESTS
assert np.array_equal(gradient(X, y, np.array([3, 10])), [ 23, 17
3]), "Function gradient is incorrect"
assert gradient(X, y, np.array([0, 10]))[0] - gradient(X, y, np.a
rray([0, 10]))[1] == 3, "Data result is incorrect"
### END HIDDEN TESTS
```

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

Expect 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 point)

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 [12]:
```
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):
        ### BEGIN SOLUTION
        grad = gradient(X, y, theta)
        theta = theta - alpha * grad
        ### END SOLUTION
        J_per_iter[iter] = cost(theta, X, y)
        gradient_per_iter[iter] = grad.T
    return (theta, J_per_iter, gradient_per_iter)
```

In [13]:
```
(theta, J_per_iter, gradient_per_iter) = gradient_descent(X, y, n
p.array([0, 10]), 0.001, 10)
print("theta:", theta)
print("J_per_iter:", J_per_iter)
print("gradient_per_iter", gradient_per_iter)
### BEGIN HIDDEN TESTS
assert np.array_equal(np.round(theta, 5), np.round([ 0.08327017,
10.02116759], 5)), "the data result in theta is incorrect"
assert np.round(gradient_per_iter[5, 0], 5) == np.round(-7.961000
25, 5), "the data result in gradient_per_iter is incorrect"
### END HIDDEN TESTS
```

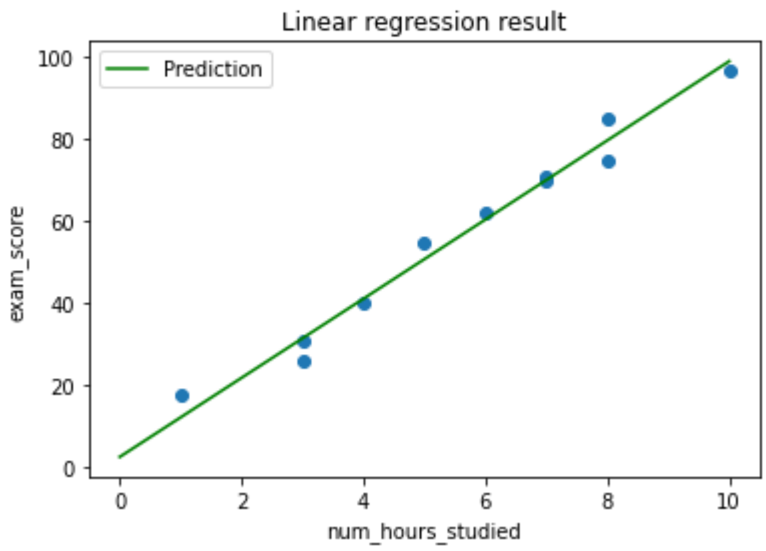
```
theta: [ 0.08327017 10.02116759]
J_per_iter: [84.775269   84.65958757 84.5793525  84.51074587 84.4
4605981 84.38279953
 84.32015717 84.25787073 84.19585485 84.13408132]
gradient_per_iter [[-10.         -13.        ]
 [ -9.084       -6.894     ]
 [ -8.556648    -3.421524  ]
 [ -8.25039038  -1.4471287 ]
 [ -8.06991411  -0.32491618]
 [ -7.96100025   0.31253312]
 [ -7.8928063    0.67422616]
 [ -7.84778746   0.87905671]
 [ -7.81596331   0.9946576 ]
 [ -7.79165648   1.05950182]]
```

Expect 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]]

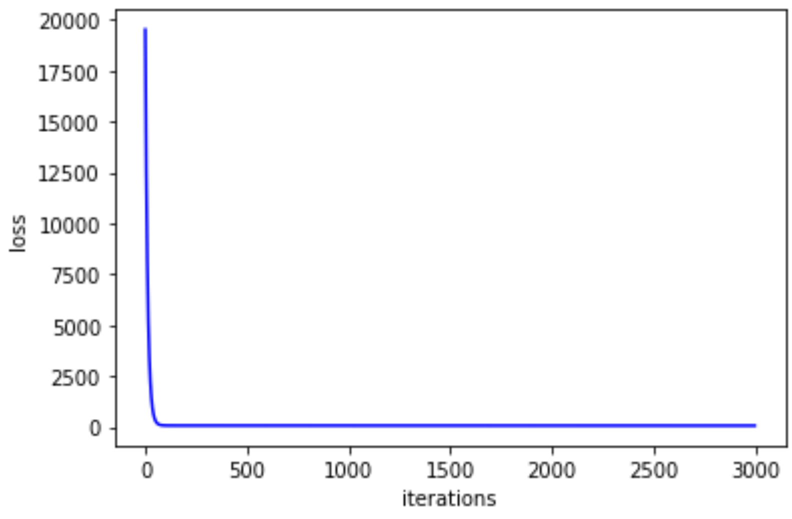
In [14]:
```
# optimize for parameters
theta_initial = np.array([0, 0])
alpha = 0.0001
iterations = 3000
theta, costs, grad = gradient_descent(X, y, theta_initial, alpha,
iterations)
print('Optimal parameters: theta_0 %f theta_1 %f' % (theta[0], th
eta[1]))
```

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

In [15]:
```
# 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 [16]:
```
# Visualize the loss
x_loss = np.arange(0, iterations, 1)

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

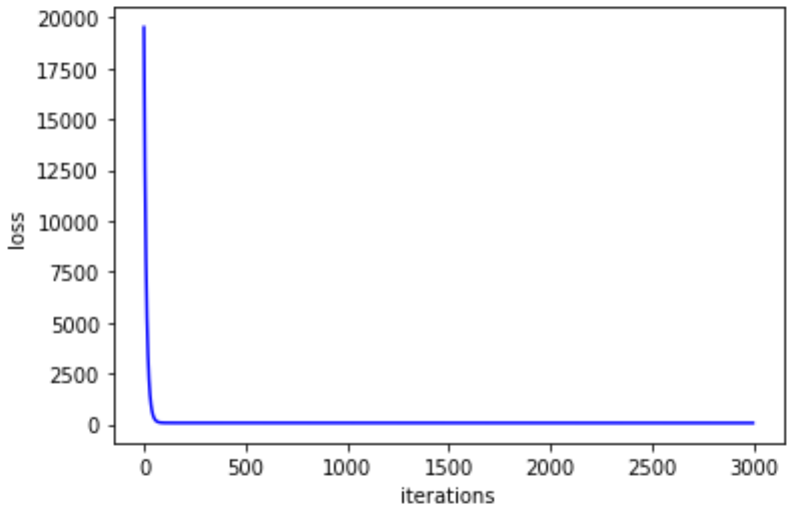


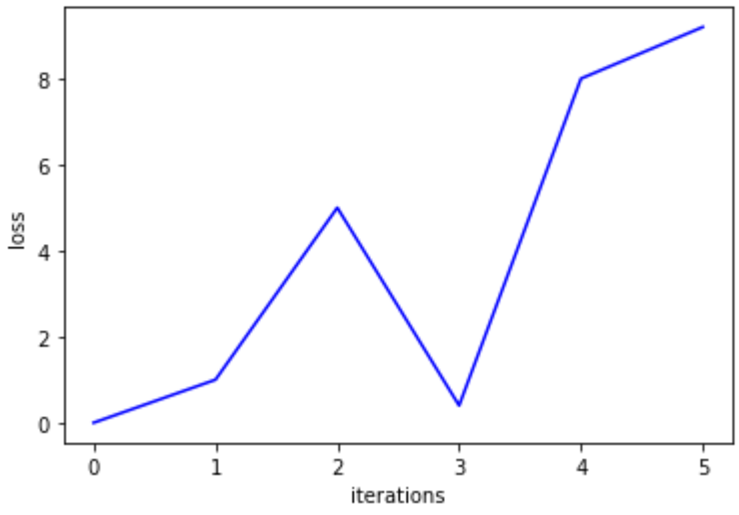
Excercise 5 (2 point)

From cost plotting graph at above, please create it as function:

In [17]:
```
def cost_plot(iterations, costs):
    ### BEGIN SOLUTION
    x_loss = np.arange(0, iterations, 1)

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

In [18]:
```
cost_plot(iterations, costs)
### BEGIN HIDDEN TESTS
from nose.tools import assert_raises
assert not cost_plot(6, np.array([0, 1, 5, 0.4, 8, 9.2])), "Funct
ion cost_plot is incorrect"
### END HIDDEN TESTS
```





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 point)

Create **goodnees of fit** function by using the equation at above

In [19]:
```
def goodness_of_fit(y, y_predicted):
    ### BEGIN SOLUTION
    r_square = 1 - np.square(y - y_predicted.T).sum()/np.square(y
- y.mean()).sum()
    ### END SOLUTION
    return r_square
```

In [20]:
```
y_predicted =  h(X, theta)
r_square = goodness_of_fit(y, y_predicted)
print(r_square)
### BEGIN HIDDEN TESTS
assert np.array_equal(np.round(r_square, 5), np.round(0.978623973
1773175, 5)), "result of r_square is incorrect"
assert np.round(r_square, 5) == np.round(0.9786239731773175, 5),
"result of r_square is incorrect"
yhat =  h(X, np.array([0, 10]))
r2 = goodness_of_fit(y, yhat)
assert np.round(r2, 5) == np.round(0.9740385950298487, 5), "Funct
ion goodness_of_fit is incorrect"
### END HIDDEN TESTS
```

```
0.9786239731773175
```

Expect 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 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 [21]:
```
# We use numpy's genfromtxt function to load the data from the te
xt file.

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

raw_data
```

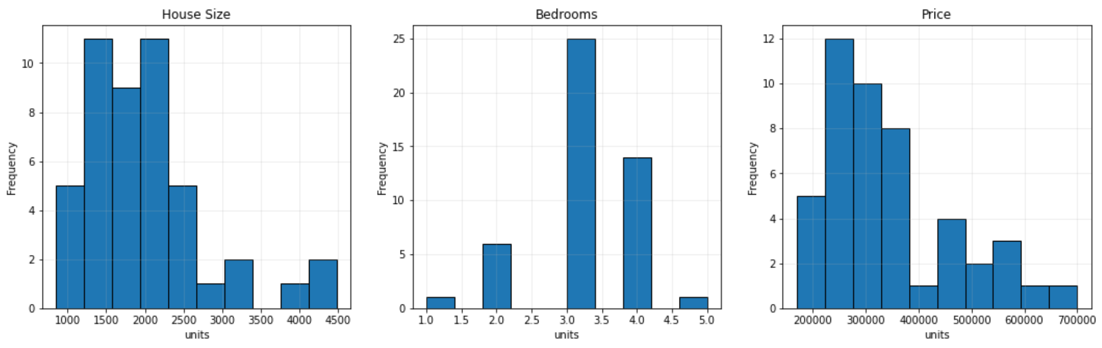
Out[21]:
```
array([['Square Feet', ' Number of bedrooms', 'Price'],
       ['2104', '3', '399900'],
       ['1600', '3', '329900'],
       ['2400', '3', '369000'],
       ['1416', '2', '232000'],
       ['3000', '4', '539900'],
       ['1985', '4', '299900'],
       ['1534', '3', '314900'],
       ['1427', '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 [22]:
```
# 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 [23]:
```
# Visualise the distribution of independent and dependent variabl
es

# 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, w
space=.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)
```



Standardization

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 will have great 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.

Standardization can help with this. For each variable, we subtract that variable's mean from every instance then divide the result by the variable's standard deviation. The result will be a set of "standardized" variables with mean 0 and variance 1.

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

In [25]:
```
# Extract y from normalized data
y_label = 'Price'
y_index = np.where(headers == y_label)[0][0]
y = np.array([data_norm[:,y_index]]).T

# Extract X from normalized data
X = data_norm[:,0:y_index]

# Insert column of 1's for intercept term
X = np.insert(X, 0, 1, axis=1)
```

In [26]:
```
# 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 point)

Optimize the parameters using gradient descent:

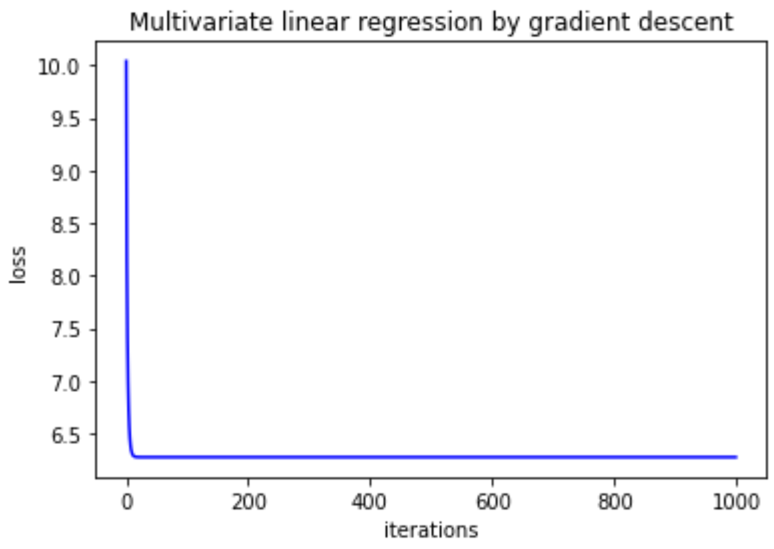
In [27]:
```
theta_initial = np.zeros((X.shape[1],1))
alpha = 0.01
iterations = 1000
### BEGIN SOLUTION
theta, costs, grad = gradient_descent(X, y, theta_initial, alpha,
iterations)
### END SOLUTION
```

In [28]:
```
print('Theta values ', theta)
### BEGIN HIDDEN TESTS
assert np.array_equal(np.round(theta, 5), np.round([[-1.00475184e
-16],[ 8.84765988e-01],[-5.31788196e-02]], 5)), "theta values are
incorrect"
a2 = 0.003
it2 = 3000
res = gradient_descent(X, y, theta_initial, a2, it2)
assert np.array_equal(np.round(res[0], 5), np.round([[-9.15933995
e-17],[8.84765988e-01],[-5.31788197e-02]], 5)), "gradient_descent
function is incorrect"
### END HIDDEN TESTS
```

```
Theta values  [[-9.65894031e-17]
 [ 8.84765988e-01]
 [-5.31788197e-02]]
```

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

In [29]:
```
# 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 point)

Compute goodness of fit

In [30]:
```
# Goodness of fit
y_predicted =  h(X,theta)
### BEGIN SOLUTION
r_square = goodness_of_fit(y, y_predicted)
### END SOLUTION
```

In [31]:
```
print(r_square)
### BEGIN HIDDEN TESTS
assert r_square == goodness_of_fit(y, h(X,theta)), "find r_square
with incorrect equation"
### END HIDDEN TESTS
```

```
-80.44841584735896
```

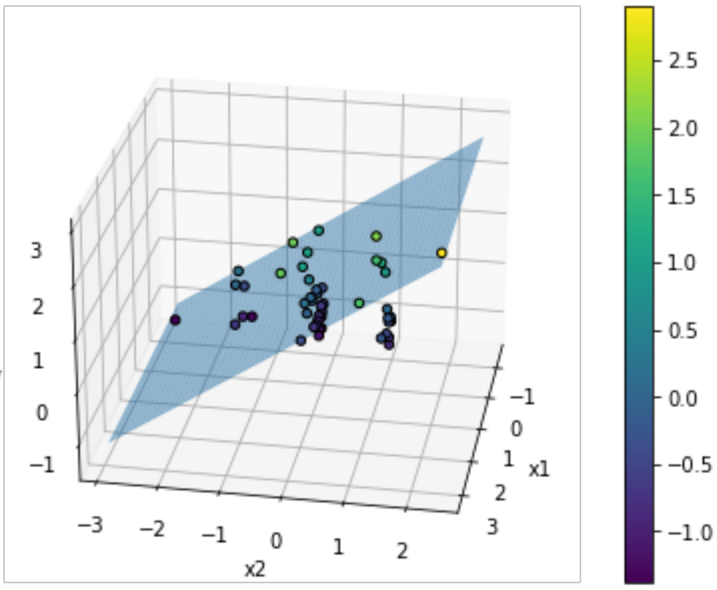
Transform standardized data back to original scale

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

In [32]:
```
# 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 [33]:
```
# 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 exercise

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 [34]:
```
# just remove all parameters
%reset
```

```
Once deleted, variables cannot be recovered. Proceed (y/[n])? y
```

In [35]:
```
import matplotlib.pyplot as plt
import numpy as np
```

Exercise 2.1 (5 point)

Download raw_data and setup data

In [36]:
```
# Download raw_data and setup data
### BEGIN SOLUTION
raw_data = np.genfromtxt('Housing_data.txt',delimiter = ',', dtyp
e=str)
headers = raw_data[0,:];
data = np.array(raw_data[1:,:], dtype=float)
### END SOLUTION
```

In [37]:
```
print(data[:5])
### BEGIN HIDDEN TESTS
assert np.array_equal(np.round(data[7], 5), np.round([1.42700e+0
3, 3.00000e+00, 1.98999e+05], 5)), "data is incorrect"
### END HIDDEN TESTS
```

```
[[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]]
```

Expect 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 point)

Normalized data

In [38]:
```
# Normalized data
def normalized_data(data):
    ### BEGIN SOLUTION
    return (data-np.mean(data, axis = 0))/np.std(data, axis = 0)
    ### END SOLUTION
```

In [39]:
```
data_norm = normalized_data(data)
print(data_norm[:5])
### BEGIN HIDDEN TESTS
assert np.array_equal(np.round(data_norm[7], 5), np.round([-0.729
68575, -0.22609337, -1.1431751 ], 5)), "data is incorrect"
### END HIDDEN TESTS
```

```
[[ 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]]
```

Expect 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 point)

Extract X and y from data

In [40]:
```
# Extract y from data
### BEGIN SOLUTION
y_label = 'Price'
y_index = np.where(headers == y_label)[0][0]
y = data_norm[:,y_index]
### END SOLUTION
```

In [41]:
```
print(y[:5])
### BEGIN HIDDEN TESTS
assert np.array_equal(np.round(y[10:14], 5), np.round([-0.8117348
5,  0.05325146, -0.08418307,  2.90606282], 5)), "data is incorrec
t"
### END HIDDEN TESTS
```

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

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

In [42]:
```
# Extract X from data
### BEGIN SOLUTION
X = data_norm[:,0:y_index]
X = np.insert(X, 0, 1, axis=1)
### END SOLUTION
```

In [43]:
```
print(X[:5,:])
### BEGIN HIDDEN TESTS
assert np.array_equal(np.round(X[10,:], 5), np.round([ 1., -0.077
1822, 1.10220517], 5)), "data is incorrect"
### END HIDDEN TESTS
```

```
[[ 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]]
```

Expect 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 point)

Create h, cost, gradient, and gradient_descent

In [44]:
```
# create h function
def h(X,theta):
    ### BEGIN SOLUTION
    y_predicted = X.dot(theta)
    ### END SOLUTION
    return y_predicted
```

In [45]:
```
print(h(X, np.array([1, 2, 4]))[:5])
### BEGIN HIDDEN TESTS
res = h(X, np.array([1, 8, 10]))
assert res.shape == (X.shape[0],), "Data size in result is incorr
ect"
# print(h(X, np.array([1, 3, 10])))
assert np.round(res[16], 5) == np.round(-21.470182911077043, 5),
"Data result is incorrect"
assert np.array_equal(np.round(h(X, np.array([1, 3, 10]))[:5],
5), np.round([-0.86668741,-2.78985577,0.26279242,-16.7749502,15.8
3526391],5)), "Function h is incorrect"
### END HIDDEN TESTS
```

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

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

In [46]:
```
def cost(theta, X, y):
    ### BEGIN SOLUTION
    y_predicted = h(X, theta) - y
    J = y_predicted.T.dot(y_predicted)/2
    ### END SOLUTION
    return J
```

In [47]:
```
print(cost(np.array([1, 8, 10]), X, y))
### BEGIN HIDDEN TESTS
assert np.round(cost(np.array([1, 8, 10]), X, y), 5) == np.round
(5477.13863, 5), "Data result is incorrect"
assert np.round(cost(np.array([2, 1, 2]), X, y), 5) == np.round(2
05.8799553398718, 5), "Function cost is incorrect"
### END HIDDEN TESTS
```

```
5477.13862837469
```

Expect result: 5477.138628374691

In [48]:
```
# Gradient of cost function
def gradient(X, y, theta):
    ### BEGIN SOLUTION
    grad = X.T.dot(h(X, theta) - y)
    ### END SOLUTION
    return grad
```

In [49]:
```
print(gradient(X, y, np.array([1, 8, 10])))
### BEGIN HIDDEN TESTS
assert np.array_equal(np.round(gradient(X, y, np.array([3.1, -2.
1, 4.8])), 5), np.round([145.7,-12.55581557,149.54496443],5)), "F
unction gradient is incorrect"
assert np.round(gradient(X, y, np.array([3.2, 1.0, 2.5]))[0] - gr
adient(X, y, np.array([3.2, 1.0, 2.5]))[1], 5) == np.round(77.788
27035558807, 5), "Data result is incorrect"
### END HIDDEN TESTS
```

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

Expect result: [47. 599.00016917 659.76139633]

In [50]:
```
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):
        ### BEGIN SOLUTION
        grad = gradient(X, y, theta)
        theta = theta - alpha * grad
        ### END SOLUTION
        J_per_iter[iter] = cost(theta, X, y)
        gradient_per_iter[iter] = grad.T
    return (theta, J_per_iter, gradient_per_iter)
```

In [51]:
```python
(theta, J_per_iter, gradient_per_iter) = gradient_descent(X, y, n
p.array([0, 1, 10]), 0.001, 10)
print("theta:", theta)
print("J_per_iter:", J_per_iter)
print("gradient_per_iter", gradient_per_iter)
### BEGIN HIDDEN TESTS
assert np.array_equal(np.round(theta, 5), np.round([-8.20787882e-
16,-7.72838948e-01,6.35294636e+00], 5)), "the data result in thet
a is incorrect"
assert np.round(gradient_per_iter[5, 0], 5) == np.round(1.1102230
2e-13, 5), "the data result in gradient_per_iter is incorrect"
### END HIDDEN TESTS
```

```
theta: [-8.48099369e-16 -7.72838948e-01  6.35294636e+00]
J_per_iter: [2123.51284628 1873.56259758 1656.90935568 1468.93187
452 1305.65834104
 1163.67477334 1040.04635308  932.24986509  838.11567544  755.777
90087]
gradient_per_iter [[1.24789068e-13 2.70000169e+02 4.75532186e+02]
 [1.12798659e-13 2.44794887e+02 4.46076185e+02]
 [1.08357767e-13 2.21549490e+02 4.18667980e+02]
 [9.10382880e-14 2.00117968e+02 3.93159744e+02]
 [8.61533067e-14 1.80365065e+02 3.69414440e+02]
 [8.29336599e-14 1.62165488e+02 3.47305031e+02]
 [6.82787160e-14 1.45403177e+02 3.26713748e+02]
 [6.53921362e-14 1.29970626e+02 3.07531415e+02]
 [5.29576383e-14 1.15768253e+02 2.89656812e+02]
 [5.54001289e-14 1.02703825e+02 2.72996100e+02]]
```

Expect 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 point)

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

In [53]:
```python
### BEGIN SOLUTION
theta_initial = np.zeros(X.shape[1])
alpha = 0.003
iterations = 30000
theta, costs, grad = gradient_descent(X, y, theta_initial, alpha,
iterations)
### END SOLUTION
```

In [54]:
```
print("theta:", theta)
print("cost_per_iter:", costs[-5:])
print("gradient_per_iter", grad[-5:])
### BEGIN HIDDEN TESTS
assert np.array_equal(np.round(theta, 5), np.round([-1.05832010e-
16,8.84765988e-01,-5.31788197e-02], 5)), "the data result in thet
a is incorrect"
assert np.round(grad[2, 1], 5) == np.round(-2.70822645e+01, 5), "
the data result in gradient_per_iter is incorrect"
### END HIDDEN TESTS
```

```
theta: [-9.83380044e-17  8.84765988e-01 -5.31788197e-02]
cost_per_iter: [6.27579208 6.27579208 6.27579208 6.27579208 6.275
79208]
gradient_per_iter [[-2.49800181e-16 -1.78188446e-14  2.09590226e-
16]
 [ 1.94289029e-16 -1.78188446e-14  1.09776865e-15]
 [-2.49800181e-16 -1.78188446e-14  2.09590226e-16]
 [ 1.94289029e-16 -1.78188446e-14  1.09776865e-15]
 [-2.49800181e-16 -1.78188446e-14  2.09590226e-16]]
```

Expect 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 point)

Calculate goodness of fit

In [55]:
```
def goodness_of_fit(y, y_predicted):
    ### BEGIN SOLUTION
    r_square = 1 - np.square(y - y_predicted.T).sum()/np.square(y
- y.mean()).sum()
    ### END SOLUTION
    return r_square
```

In [56]:
```
y_predicted =  h(X, theta)
r_square = goodness_of_fit(y, y_predicted)
print(r_square)
### BEGIN HIDDEN TESTS
assert np.array_equal(np.round(r_square, 5), np.round(0.732945018
0289143, 5)), "result of r_square is incorrect"
assert np.round(r_square, 5) == np.round(0.7329450180289143, 5),
"result of r_square is incorrect"
yhat =  h(X, np.array([0, 1, 10]))
r2 = goodness_of_fit(y, yhat)
# print(r2)
assert np.round(r2, 5) == np.round(-101.6441465600189, 5), "Funct
ion goodness_of_fit is incorrect"
### END HIDDEN TESTS
```

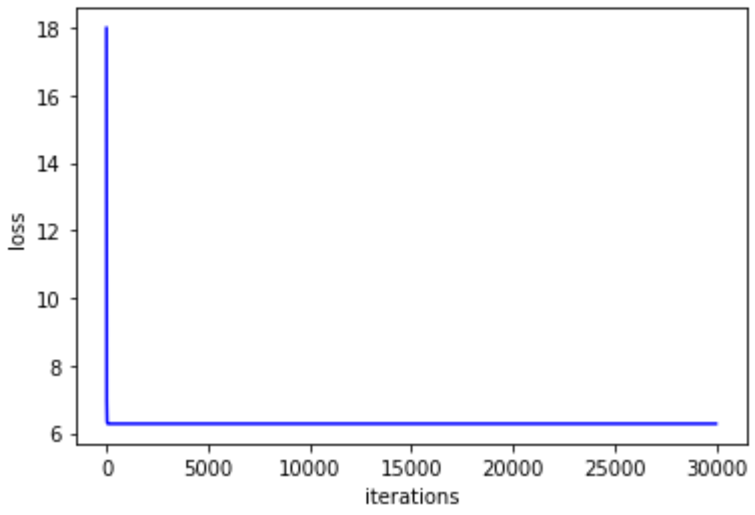
```
0.7329450180289143
```

Expect result: 0.7329450180289143

Excercise 2.7 (2 point)

Plot graph of cost results

In [59]:
```
def cost_plot(iterations, costs):
    ### BEGIN SOLUTION
    x_loss = np.arange(0, iterations, 1)

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

cost_plot(iterations, costs)
```



Exercise 2.8 (8 point)

Write the function of **normal equation** and write normal equation code:

Write Normal equation here!

```
In [60]: # write normal equation code
         def normal_equation(X,y):
             ### BEGIN SOLUTION
             X_sq = np.linalg.inv(X.T.dot(X))
             XtY = X.T.dot(y)
             theta = (X_sq.dot(XtY))
             ### END SOLUTION
             return theta
```

```
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)
         ### BEGIN HIDDEN TESTS
         assert np.array_equal(np.round(theta_norm.T, 5), np.round([[-7.90
         434550e-17,8.84765988e-01,-5.31788197e-02]], 5)), "the data resul
         t in theta is incorrect"
         assert np.array_equal(np.round(r_norm_square, 5), np.round(0.7329
         450180289143, 5)), "result of r_square is incorrect"
         assert np.round(r_norm_square, 5) == np.round(0.7329450180289143,
         5), "result of r_square is incorrect"
         ### END HIDDEN TESTS
```

```
theta from normal equation: [[-7.76616596e-17  8.84765988e-01 -5.
31788197e-02]]
r_square: 0.7329450180289143
```

Expect 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?

Explaination here.

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

In [63]:
```
# Write code here and below
```

In [ ]:
```
```