Assignment 1 (A): Building a Linear Regression Algorithm with Application to Used Car Price Prediction

Acknowledgment

You are required to acknowledge the following statement by entering your full name, SID, and date below:

"By continuing to work on or submit this deliverable, I acknowledge that my submission is entirely my independent original work done exclusively for this assessment item. I agree to

- Submit only my independent original work
- Not share answers and content of this assessment with others
- Report suspected violations to the instructor

Furthermore, I acknowledge that I have not engaged and will not engage in any activities that dishonestly improve my results or dishonestly improve/hurt the results of others, and that I abide to all academic honor codes set by the University."

Your full name:
Pang Fong Chun
Your SID:
3035281299
Date:
11th June 2022

1. Introduction

In this part of the assignment, you will implement the linear regression learning algorithm and apply it to predicting prices of used cars. You are required to complete the lines between **START YOUR CODE HERE** and **END YOUR CODE HERE** (if applicable) and to execute each cell. Within each coding block, you are required to enter your code to replace `None` after the `=` sign (except otherwise stated). You are not allowed to use other libraries or files than those provided in this assignment. When entering your code, you should not change the names of variables, constants, and functions already listed.

Contents

In [1]:

```
# You need to import the libraries required for this programming exercise.
# Scientific and vector computation for python
import numpy as np
# Data analysis and manipulation tool for python
import pandas as pd

# Plotting library
import matplotlib.pyplot as plt

# tells matplotlib to embed plots within the notebook
%matplotlib inline
```

2. Used Car Dataset

2.1. Data Description

The dataset includes 5,996 records of used cars. Each record is described by 12 features as listed below (an additional unnamed ID (first column) is not listed). The text file named `raw_regression_data.csv` stores each record as one row having the feature values separated by commas.

Feature	Description
Location	Country of the car
Vehicle_Year	Age (in years) of the car
Kilometers_Driven	Distance (in km) traveled by the used car to date
Fuel_Type	Type of fuel used by the car
Transmission	Type of the transmission

Feature	Description
Owner_Type	Type of the owner
Seats	Number of seats in a used car
Company	Vehicle make of the used car
Fuel_Consumption(kmpl)	Fuel consumption per liter
Engine(CC)	Swept volume (Displacement of one cylinder)
Power(bhp)	Brake horse power (bhp) is the unit of power of an engine without any losses like heat and noise
Price	Selling price of the used car

2.2. Data Loading

In this section, you use the pandas functions `read_csv` to load the dataset, `info()` to generate a summary, `drop()` to drop the first unnamed feature. You can optionally use `head()` to display first several records.

In [2]:

```
# read in the data
raw_regression_data = pd.read_csv('raw_regression_data.csv')
raw_regression_data.drop('Unnamed: 0', axis=1, inplace=True)
raw_regression_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5996 entries, 0 to 5995
Data columns (total 12 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   Location                5996 non-null   object 
 1   Vehicle_Year            5996 non-null   int64  
 2   Kilometers_Driven       5996 non-null   int64  
 3   Fuel_Type               5996 non-null   object 
 4   Transmission            5996 non-null   object 
 5   Owner_Type              5996 non-null   object 
 6   Seats                   5996 non-null   float64
 7   Company                 5996 non-null   object 
 8   Fuel_Consumption(kmpl)  5996 non-null   float64
 9   Engine(CC)              5996 non-null   float64
 10  Power(bhp)              5996 non-null   float64
 11  Price                   5996 non-null   float64
dtypes: float64(5), int64(2), object(5)
memory usage: 562.2+ KB
```

In [3]:

```
print(raw_regression_data.head())
```

```
     Location  Vehicle_Year  Kilometers_Driven   Fuel_Type Transmission  \
0      Mumbai            10              72000  Clean_Fuel       Manual   
1        Pune             5              41000      Diesel       Manual   
2     Chennai             9              46000      Petrol       Manual   
3     Chennai             8              87000      Diesel       Manual   
4  Coimbatore             7              40670      Diesel    Automatic   

  Owner_Type  Seats  Company  Fuel_Consumption(kmpl)  Engine(CC)  Power(bhp)  
\
0      First    5.0   MARUTI                   26.60       998.0       58.16   
1      First    5.0  HYUNDAI                   19.67      1582.0      126.20   
2      First    5.0    HONDA                   18.20      1199.0       88.70   
3      First    7.0   MARUTI                   20.77      1248.0       88.76   
4     Second    5.0     AUDI                   15.20      1968.0      140.80   

   Price
```

```
0     1.75
1    12.50
2     4.50
3     6.00
4    17.74
```

2.3. Data Visualization

You can visualize the distribution of each feature by executing the following code block. All numeric (continuous) features are visualized by blue bars, whereas all categorical features are visualized by red bars.

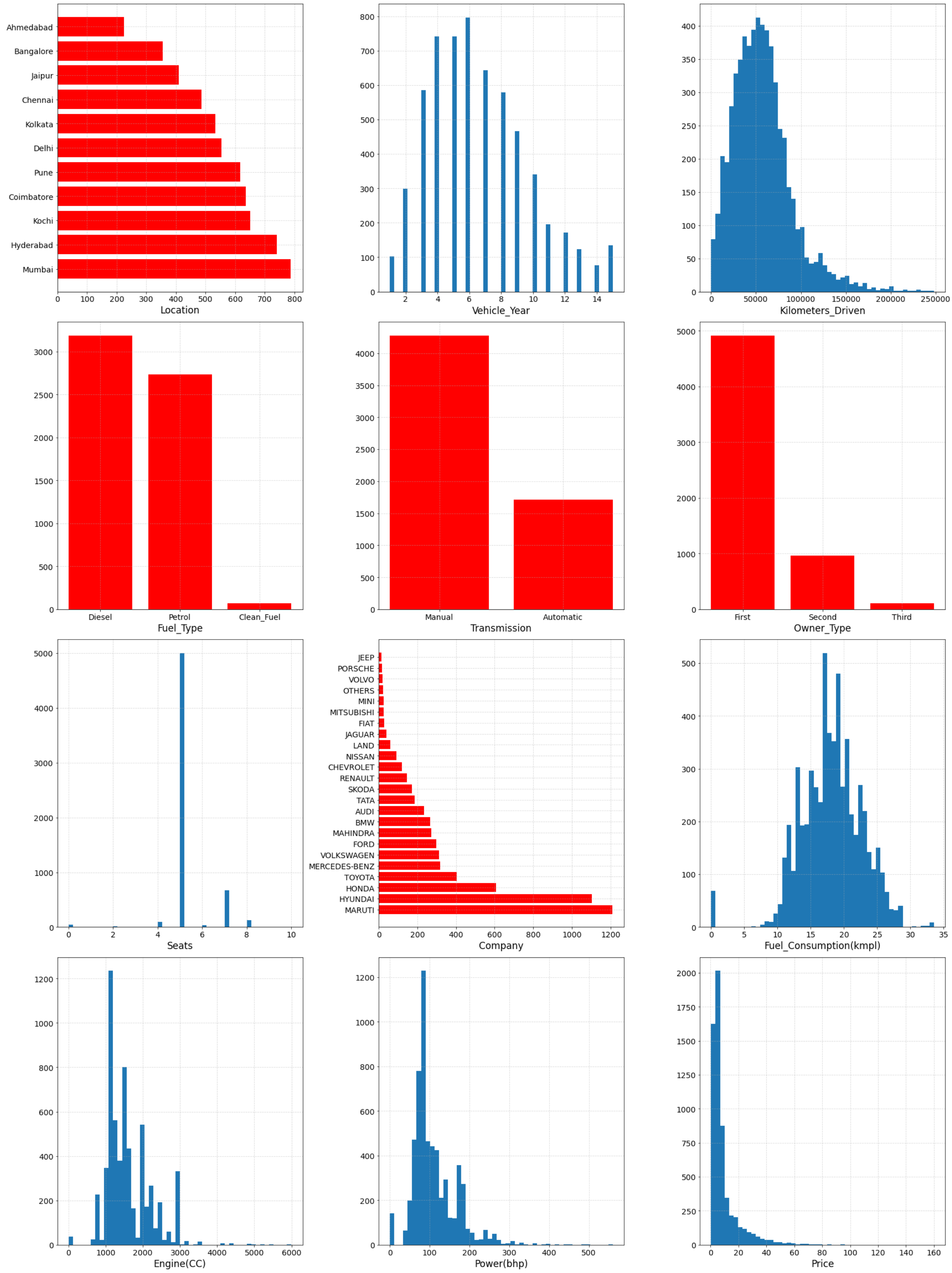
In [4]:

```
attribute_number = len(raw_regression_data.columns)
print("Attribute Number: {}".format(attribute_number))

# subplots
fig = plt.figure(figsize=(24, 32))
ax = fig.subplots(attribute_number//3,3)
# iterations
for num, title in enumerate(raw_regression_data.columns):
    idx = num//3 # divided with no remainder
    idy = num%3  # remainder
    if raw_regression_data[title].dtype in ['object']:
        value_count_dict = raw_regression_data[title].value_counts().to_dict(
        keys = list(value_count_dict.keys())
        values = list(value_count_dict.values())
        if len(raw_regression_data[title].unique().tolist()) < 8:
            ax[idx, idy].bar(keys, values, color='r')
        else:
            ax[idx, idy].barh(keys, values, color='r')
    else:
        ax[idx, idy].hist(raw_regression_data[title].values, bins=50);

    # set title with attribute
    ax[idx, idy].set_xlabel(title, fontsize=17)
    # set grid width
    ax[idx, idy].grid(linestyle='--', alpha=0.5)
    # font size of ticks
    ax[idx, idy].tick_params(labelsize=14)
plt.tight_layout()
```

```
Attribute Number: 12
```



2.4. One-hot Encoding

All categorial data (e.g., fuel type) must be transformed into numerical indices. You will use the function `get_dummies()` from the Pandas library to perform this one-hot encoding.

In [5]:
```
# one-hot encoding
regression_data = pd.get_dummies(raw_regression_data)
print('Before using get_dummies\nFuel_Type: {}'.format(raw_regression_data.lo

print('\nAfter using get_dummies:')
print('Fuel_Type_Clean_Fuel: {}'.format(regression_data.loc[0, 'Fuel_Type_Cle
```

```
print('Fuel_Type_Diesel: {}'.format(regression_data.loc[0, 'Fuel_Type_Diesel'
print('Fuel_Type_Petrol: {}'.format(regression_data.loc[0, 'Fuel_Type_Petrol'
```

```
Before using get_dummies
Fuel_Type: Clean_Fuel

After using get_dummies:
Fuel_Type_Clean_Fuel: 1
Fuel_Type_Diesel: 0
Fuel_Type_Petrol: 0
```

2.5. Feature Scaling

You will implement two feature scaling techniques, `min_max_scaler()` and `z_score_scaler()` , to normalize the input values to ensure efficient convergence of the algorithm.

2.5.1. Min-max scaling

Task 1: The min-max scaling equation is defined as follows:

$$\text{min\_max\_scaler}(x_i) = \frac{x_i - x_{min}}{x_{max} - x_{min}} \,. \tag{1}$$

In the function `min_max_scaler()` , if x_{min} and x_{max} are not given as inputs, the minimal and maximal values per features can be found by using `np.min()` and `np.max()` functions. To compute the scaled `new_x` , you need to use `np.divide()` by setting its first and second parameters to numerator and denominator of the equation above. To avoid the problem of division by zero, you need to set the `out` parameter of `np.divide()` by using `np.zeros_like()` (enter the numerator as its parameter). You also need to set the `where` parameter to indicate the condition, e.g., if the denominator is named `denom` , the condition is `denom!=0` .

In [9]:
```
# Min-max range normalization
def min_max_scaler(x, x_min=None, x_max=None):
    """
        feature scaling with min-max range normalization
        x : arrary_like
            dataset with several features
        x_min : float
            given maximal value of features. If this input are given, the dat
            If not, this value will be calculated by the data themselves.
        x_max : float
            given minimal value of features. If this input are given, the dat
    """

    new_x = np.zeros_like(x) # create a new matrix "new_x" with the shape as

    # Task 1:
    # ====================== START YOUR CODE HERE ======================
    # check if the necessary minimum and maximum are given
    # 1. minimum value per feature element (column) (1 line code)
    # 2. maximum value per feature element (column) (1 line code)
    # 3. division considering zero denominator (1 line code)
    if x_min is None or x_max is None: # Please do not change this line !!!
        x_min = x.min(axis=0)
        x_max = x.max(axis=0)
    x_range = x_max-x_min
    new_x = np.divide(x-x_min,x_range,out=np.zeros_like(x-x_min),where=x_rang
```

```
    # ====================== END YOUR CODE HERE =========================

    return new_x, x_min, x_max
```

Inverse min-max scaling

To recover the original data, the following function `inverse_min_max_scaler()` can be used.

In [10]:
```
# inverse min-max scaling
def inverse_min_max_scaler(scaled_x, parameters):
    """
        inverse feature scaling of the min-max normalization
    """
    x_min, x_max = parameters
    return scaled_x * (x_max-x_min) + x_min
```

[Test Block 1]: Test code for function `min_max_scaler()` . First 10 data items are extracted from dataset. Only two features are of interest.

In [11]:
```
# features of interest (two features)
demo_features = ['Vehicle_Year', 'Kilometers_Driven']
# sample the first ten data items
data_sample = regression_data[demo_features].head(10).values.astype('float')
# implemented function
scaled_sample, sample_min, sample_max = min_max_scaler(data_sample)

print('Minimal Value: {}'.format(sample_min))
print('Maximal Value: {}'.format(sample_max))
# you can use function "np.allclose" to compare two floats with small differe
if np.allclose(sample_min, [4.0, 36000.0]) and np.allclose(sample_max, [10.0,
   and np.allclose(scaled_sample[[0, -1],1], [0.70588235, 0.58690196]):
    print('Your answers are correct!')
else:
    print('Your answers are not correct, please correct the function codes.')
```

```
Minimal Value: [4.0e+00 3.6e+04]
Maximal Value: [1.0e+01 8.7e+04]
Your answers are correct!
```

2.5.2. Z-score scaling

Task 2: The z-score scaling equation is defined as follows:

$$\text{z\_score\_scaler}(x_i) = \frac{x_i - \bar{x}_i}{s_{x_i}} .$$

The function `z_score_scaler()` transfroms the original data distribution to a normal distribution with zero mean and one standard variation. You should use `np.mean()` and `np.std()` to get mean value $\bar{x}_i$ and standard deviation s_{x_i} respectively. Then, you should use `np.divide()` to compute the scaled `new_x` and set the `out` parameter using `np.zeros_like()` (enter the numerator as its parameter) and set the `where` parameter to indicate the condition, e.g., if the denominator is named `denom` , the condition is `denom!=0` .

In [12]:
```
# Z-score normalization
def z_score_scaler(x, x_mean=None, x_std=None):
```

```
    """
        feature scaling with Z-score normalization (standarlization)
    """

    new_x = np.zeros_like(x) # create a new matrix "new_x" with the shape as

    # Task 2:
    # ====================== START YOUR CODE HERE ======================
    # check if the necessary mean and standard variation values is given
    # 1. mean value per feature element (column) (1 line code)
    # 2. standard deviation per feature element (column) (1 line code)
    # 3. division considering zero denominator (1 line code)
    if x_mean is None or x_std is None: # Please do not change this line !!!
        x_mean = x.mean(axis=0)
        x_std = x.std(axis=0)
    new_x = np.divide(x-x_mean,x_std,out=np.zeros_like(x-x_mean),where=x_std!
    # ====================== END YOUR CODE HERE ======================

    return new_x, x_mean, x_std
```

Inverse z-score scaling

To recover the original data, you can use the `inverse_z_score_scaler()` function.

In [13]:

```
# inverse z-score scaler
def inverse_z_score_scaler(scaled_x, parameters):
    """
        inverse feature scaling with Z-score normalization (standarlization)
    """
    x_mean, x_std = parameters
    return scaled_x * x_std + x_mean
```

[Test Block 2]: Test code for function `z_score_scaler()` . First 10 data items are extracted from dataset. Only two features are of interest.

In [14]:

```
# target data
demo_features = ['Vehicle_Year', 'Kilometers_Driven']
# sample the first ten data items
data_sample = regression_data[demo_features].head(10).values
scaled_sample, sample_mean, sample_std = z_score_scaler(data_sample.astype('f

print('Mean Value: {}'.format(sample_mean))
print('Standard Variation: {}'.format(sample_std))
# you can use function "np.allclose" to compare two floats with small differe
if np.allclose(sample_mean, [7.3, 61503.1]) and np.allclose(sample_std, [1.67
    and np.allclose(scaled_sample[[0, -1],1], [0.57215793, 0.24140749]):
    print('Your answers are correct!')
else:
    print('Your answers are not correct, please correct the funtion codes.')
```

```
Mean Value: [7.30000e+00 6.15031e+04]
Standard Variation: [1.67630546e+00 1.83461585e+04]
Your answers are correct!
```

2.5.3. Training-testing dataset scaling

You will execute the following code to scale the feature values using a selected method.

In [15]:

```
# feature scaling
def scale_feature(x_train, x_test, method='min_max'):
```

```
    """
        sacling the features in training and testing dataset
        only with distribution of training dataset.
    """

    scaled_train_data = np.zeros_like(x_train)
    scaled_test_data = np.zeros_like(x_test)

    if method == 'min_max':
        scaled_train_data, train_x_min, train_x_max = min_max_scaler(x_train)
        scaled_test_data, train_x_min, train_x_max = min_max_scaler(x_test, t
        parameters = (train_x_min, train_x_max)
    elif method == 'z_score':
        scaled_train_data, train_x_mean, train_x_std = z_score_scaler(x_train
        scaled_test_data, train_x_mean, train_x_std = z_score_scaler(x_test, 
        parameters = (train_x_mean, train_x_std)
    else:
        raise ValueError("The mentioned method have not been implemented yet,
                         please select one from min-max and z-score normaliza

    return scaled_train_data, scaled_test_data, parameters
```

To recover the original data, you can use the `inverse_scale_feature()` function using a selected method.

In [16]:

```
# inverse feature scaling
def inverse_scale_feature(scaled_x, parameters, method='min_max'):
    """
        inverse sacling the features in training and testing dataset
        only with distribution of training dataset.
    """

    data_x = np.zeros_like(scaled_x)

    if method == 'min_max':
        data_x = inverse_min_max_scaler(scaled_x, parameters)
    elif method == 'z_score':
        data_x = inverse_z_score_scaler(scaled_x, parameters)
    else:
        raise ValueError("The mentioned method have not been implemented yet,
                        please select one from min-max and z-score normalizat

    return data_x
```

2.6. Train-test Split

Task 3:

You will implement `train_test_split()` to split the original dataset into training and testing sets. To select the m data items randomly, you can use `np.random.permutation()` to get a random permutation of m indices.

In [17]:

```
# train-test dataset split
def train_test_split(x, y, train_ratio=0.8):
    """
        Separate the dataset into training and testing dataset for learning a
        of linear regression.

        Parameters
        ----------
```

```
        x : array_like, the input dataset of shape (m, n+1).
        y : array_like, value at given features. A vector of shape (m, 1).

        train_size: float, the percetage of training dataset (between 0 and 1

        Returns
        -------
        x_train : array_like, matrix of the training dataset.
        x_test : array_like, matrix of the testing dataset.
        y_train : array_like, value at given features in training datset. A v
        y_test : array_like, value at given features in testing dataset. A ve
    """

    m = x.shape[0]
    # Task 3:
    # ====================== START YOUR CODE HERE ======================
    # your task is:
    # 1. shuffle indices with random order (1 line code)
    # 2. multiply train_ratio and the size of dataset; then cast the result a
    row_indices = np.random.permutation(m)
    training_set_num = int(train_ratio*m)
    # ====================== END YOUR CODE HERE ======================

    # Create a Training Set
    x_train = x[row_indices[:training_set_num],:]
    y_train = y[row_indices[:training_set_num],:]

    # Create a Test Set
    x_test = x[row_indices[training_set_num:],:]
    y_test = y[row_indices[training_set_num:],:]

    return x_train, x_test, y_train, y_test
```

[Test Block 3]: Test code for function `train_test_split()` . First 100 data items are extracted from dataset. 85% of dataset will be extracted as training dataset, while the rest is in testing set.

In [18]:

```
# sample the first ten data items
label_name = 'Price'
feature_name = list(regression_data.columns)
feature_name.remove(label_name)

data_sample = regression_data.head(100)
data_sample_x = data_sample.loc[:, feature_name].values
data_sample_y = np.atleast_2d(data_sample.loc[:, label_name].values).T

(x_sample_train, x_sample_test, \
y_sample_train, y_sample_test)= train_test_split(data_sample_x, data_sample_y

# number of data items of whole dataset, training set, and testing set
data_size = data_sample.shape[0]
train_size = x_sample_train.shape[0]
test_size = x_sample_test.shape[0]
# print(train_size, test_size, data_size)
if train_size == 0.85*data_size and test_size == 0.15*data_size:
    print('Your answers are correct!')
else:
    print('Your answers are not correct, please correct the funtion codes.')
```

```
Your answers are correct!
```

2.7. Data Processing

Now, you will use your implemented functions `scale_feature()` and `train_test_split()` to process the original dataset.

Task 4:

- Separate the dataset into training (85%) and testing (15%) dataset with `train_test_split()` (1 line code)
- Processing training and testing data with feature scaling methods with `scale_feature()` , please use **Min-max** scaler for further operations. (1 line code)

In [19]:

```
# Here you should do necessary operations for the regression data.
data = regression_data.loc[:, list(regression_data.columns)[1:-1]]
data.drop('Price', axis=1, inplace=True)
data_x = data.values
data_y = np.atleast_2d(regression_data['Price'].values).T

# Task 4:
# ====================== START YOUR CODE HERE ======================
# your task here is:
# 1. train test split (1 line code)
# 2. feature scaling for training and testing dataset (1 line code)
x_train, x_test, y_train, y_test = train_test_split(data_x, data_y, train_rat
x_train, x_test, scaling_parameters = scale_feature(x_train, x_test, method='
# ====================== END YOUR CODE HERE ========================

x_train = np.concatenate([np.ones((x_train.shape[0], 1)), x_train], axis=1)
x_test = np.concatenate([np.ones((x_test.shape[0], 1)), x_test], axis=1)
y_train = np.log(y_train)
y_test = np.log(y_test)
```

3. Linear Regression Learning Algorithm

3.1. Hypothesis

The linear regression hypothesis is represented as follows:

$$h_\theta(x) = \theta^T x \ . \tag{3}$$

Task 5:

You will implement the linear regression hypothesis function as in `hypothesis()` . You can use `np.matmul()` or `np.dot()` to perform matrix multiplication.

In [20]:

```
# hypothesis function with linear model
def hypothesis(theta, x):
    """
        Hypothesis function with linear model.
        with parameters theta for linear regression and data points in x.

        Parameters
        ----------
        theta: array_like
            The parameters for the regression function. This is a vector of
            shape (n+1, 1).

        x : array_like
            The input dataset of shape (m, n+1), where m is the number of exa
```

```
            and n is the number of features. Assume that a vector of one's al
            appended to the features so the n+1 columns are given.

        Returns
        -------
        h : array_like
            Predicted values at given features. A vector of shape (m, 1).
    """

    h = np.zeros((x.shape[0],))

    # task 5:
    # ====================== START YOUR CODE HERE ======================
    # multiplication between matrix x and vector theta (1 line code)
    h = np.matmul(x, theta)
    # ====================== END YOUR CODE HERE ======================

    return h
```

[Test Block 4]: Test code for function `hypothesis()` .

In [21]:

```
demo_theta = np.array([[1, 2, 3]]).T
print('Shape of theta: {}'.format(demo_theta.shape))
demo_x = np.array([[1, 2, 3], [4, 5, 6]])
print('Shape of x: {}'.format(demo_x.shape))

h = hypothesis(demo_theta, demo_x)
print("Hypothesis value: {}".format(h))

if np.allclose(h, [[14], [32]]):
    print('Your answers are correct!')
else:
    print('Your answers are not correct, please correct the funtion codes.')
```

```
Shape of theta: (3, 1)
Shape of x: (2, 3)
Hypothesis value: [[14]
 [32]]
Your answers are correct!
```

3.2. Cost Function

3.2.1. Cost fucntion without regularization

The objective of linear regression (without the regularization term) is to search for the optimal parameters θ to minimize this cost function:

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)^2 \tag{4}$$

Task 6: In this task, you will:

1. compute the hypothesis value `hyp` with your implemented function `hypothesis()` (1 line)
2. compute the error between `hyp` and input y with function `np.substract()` (1 line)
3. compute the squared error with `np.power()` (1 line)
4. compute the cost value $J(\theta)$ with `np.sum()` (1 line)

In [22]:

```python
# Cost function without regularization term
def cost_computation(theta, x, y):
    """
    Cost function for linear regression. Computes the cost of using theta as
    parameter for linear regression to fit the data points in x and y.

    Parameters
    ----------
    theta : array_like
        The parameters for the regression function. This is a vector of
        shape (n+1, 1).

    x : array_like
        The input dataset of shape (m, n+1), where m is the number of example
        and n is the number of features. Assume a vector of one's already
        appended to the features so the n+1 columns are given.

    y : array_like
        The values of the function at each data point. This is a vector of
        shape (m, 1).

    Returns
    -------
    cost : float
        The value of cost function.

    Instructions
    ------------
    Compute the cost of a particular choice of theta and return it.
    """

    m = x.shape[0]
    cost = .0

    # Task 6:
    # ====================== START YOUR CODE HERE ======================
    # your task is:
    # 1. compute the hypothesis value (1 line code)
    # 2. compute the error between hypothesis and y with np.substract (1 line
    # 3. compute the squared error (np.power) (1 line code)
    # 4. compute the cost value (np.sum) (1 line code)
    hyp = hypothesis(theta, x)
    errors = np.subtract(hyp, y)
    squared_errors = np.power(errors, 2)
    cost = np.divide(1,2*m,where=m!=0)*np.sum(squared_errors)
    # ====================== END YOUR CODE HERE ======================

    return cost
```

[Test Block 5]: Test code for function `cost_computation()` .

In [23]:

```python
# small demo for verification
demo_theta = np.array([1, 2, 3], ndmin=2).T # shape (3, 1)
demo_x = np.array([1, 2, 3], ndmin=2)
demo_y = 20

cost_value = cost_computation(demo_theta, demo_x, demo_y)
print('Cost value: {}'.format(cost_value))

if cost_value == 18.0:
    print('Your answers are correct!')
else:
    print('Your answers are not correct, please correct the funtion codes.')
```

```
Cost value: 18.0
Your answers are correct!
```

3.2.2. Cost function with regularization

Adding a regularization term, the objective of linear regression has a slightly different cost function than (4):

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right)^2 + \frac{\lambda}{2m} \sum_{j=1}^{n} \theta_j^2 \,. \tag{5}$$

Equation (5) uses a hyperparameter λ (a positive number) that controls the values of parameters θ while the cost is being minimized. The higher the value of λ is, the lower the values of parameters θ have to be in order to minimize the cost (and vice versa).

Task 7: Your task is to:

1. compute the hypothesis value `hyp` with your implemented function `hypothesis()` (1 line)
2. compute the error between hypothesis and y with `np.substract()` (1 line)
3. compute the squared error (`np.power()`) (1 line)
4. compute the cost value (`np.sum()`) (1 line)
5. compute the regularized cost value with `np.dot()` or `np.matmul()` . Note that the output of `np.dot()` is an `np.ndarry` (shape=(1,1)). To obtain a scalar value, you need to use the method `item()` . (1 line)

In [29]:

```python
# cost function with regularization term
def regularized_cost_computation(theta, x, y, lamda):
    """
    Cost function for linear regression with a regularization term. Computes 
    parameter for linear regression to fit the data points in x and y.

    Parameters
    ----------
    theta : array_like
        The parameters for the regression function. This is a vector of
        shape (n+1, 1).

    x : array_like
        The input dataset of shape (m, n+1), where m is the number of example
        and n is the number of features. Assume that a vector of one's alread
        appended to the features so n+1 columns are given.

    y : array_like
        The values of the function at each data point. This is a vector of
        shape (m, 1).

    lamda : float
        Hyperparameter for regularization term.

    Returns
    -------
    cost : float
        The value of cost function.

    Instructions
    ------------
```

```
    Compute the cost of a particular choice of theta and return it.
    """

    m = x.shape[0]
    cost = .0

    # Task 7:
    # ====================== START YOUR CODE HERE ======================
    # 1. compute the hypothesis value (1 line code)
    # 2. compute the error between hypothesis and y with np.substract "errors
    # 3. compute the squared error "squared_errors" (np.power) (1 line code)
    # 4. compute the cost value "error_cost" (np.sum) (1 line code)
    # 5. compute the regularization cost value "regularization_cost" (1 line 
    hyp = hypothesis(theta, x)
    errors = np.subtract(hyp, y)
    squared_errors = np.power(errors, 2)
    error_cost = np.divide(1,2*m,where=m!=0)*np.sum(squared_errors)
    regularization_cost = np.divide(lamda,2*m,where=m!=0) * np.sum(np.power(t
    # regularization_cost = np.divide(lamda,2*m,where=m!=0) * np.sum(np.matmu
    # ====================== END YOUR CODE HERE ======================
    cost = error_cost + regularization_cost

    return cost
```

[Test Block 6]: Test code for function `regularized_cost_computation()` .

In [30]:

```
# small demo for verification
demo_theta = np.array([1, 2, 3], ndmin=2).T # shape (3, 1)
demo_x = np.array([1, 2, 3], ndmin=2)
demo_y = 20
lamda = 1

cost_value = regularized_cost_computation(demo_theta, demo_x, demo_y, lamda)
print('Cost value: {}'.format(cost_value))

if cost_value == 25.0:
    print('Your answers are correct!')
else:
    print('Your answers are not correct, please correct the funtion codes.')
```

```
Cost value: 25.0
Your answers are correct!
```

3.3. Gradient Descent

Next, you will implement the gradient descent algorithm to find the θ of the optimal linear regression hypothesis (or model).

3.3.1. Gradient descent without regularization

The equation to compute for parameter update (without using regularization) is:

$$\theta_j = \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)} \right) x_j^{(i)} \qquad \text{simultaneously update } \theta_j \text{ for all } j \in \{1,$$

The gradient descent algorithm iteratively reduces the cost $J(\theta)$ by find parameters θ_j by searching among all values of available features.

Task 8: In this task, you will:

1. compute the hypothesis value with your implemented function `hypothesis()` saved in "hyp" (1 line)
2. compute the difference between "hyp" and input y with function `np.substract()` , then save it in "hyp_diff" (1 line code)
3. compute the element-wise multiplication between "hyp_diff" and the j-th column of x with function `np.multiply()` , then saved into "error_list" (1 line)
4. compute the sum of errors with `np.sum()` , saved into "total_error" (1 line)
5. update each element of theta according to the equation (6). (1 line)

In [31]:

```
# update theta with gradient descent (one iteration)
def gradient_descent(theta, x, y, alpha):
    """
    Performs gradient descent to learn `theta`. Updates theta with only one i
    i.e., one gradient step with learning rate `alpha`.

    Parameters
    ----------
    theta : array_like
        Initial values for the linear regression parameters.
        A vector of shape (n+1, 1).

    x : array_like
        The input dataset of shape (m, n+1).

    y : array_like
        Value at given features. A vector of shape (m, 1).

    alpha : float
        The learning rate.

    Returns
    -------
    theta : array_like
        The learned linear regression parameters. A vector of shape (n+1, 1).

    cost : float
        cost value with respect to the current vector theta.

    Instructions
    ------------
    Peform a single gradient step on the parameter vector theta.
    """

    # Initialize some useful values
    m = y.shape[0]
    n = theta.shape[0]
    new_theta = np.zeros((n, 1))

    # Task 8:
    # ====================== START YOUR CODE HERE ======================
    hyp = hypothesis(theta, x)
    hyp_diff = np.subtract(hyp, y)
    for j in range(n):
        x_column = np.reshape(x[:, j], (-1, 1)) # make sure this is a 2D arra
        error_list = np.multiply(hyp_diff, x_column)
        total_error = np.sum(error_list)
        new_theta[j] = np.subtract(theta[j], np.divide(alpha,m,where=m!=0)*to
    # ====================== END YOUR CODE HERE ======================
```

```
    return new_theta
```

[Test Block 7]: Test code for function `gradient_descent()` .

In [32]:

```
# small demo for verification
demo_theta = np.array([1, 2, 3], ndmin=2).T # shape (3, 1)
demo_x = np.array([1, 2, 3], ndmin=2)
demo_y = np.array([20])
alpha = 0.1

new_theta = gradient_descent(demo_theta, demo_x, demo_y, alpha)
print('Updated theta value: [{}, {}, {}]'.format(new_theta[0], new_theta[1], 

if np.allclose(new_theta, np.array([[1.6], [3.2], [4.8]])):
    print('Your answers are correct!')
else:
    print('Your answers are not correct, please correct the funtion codes.')
```

```
Updated theta value: [[1.6], [3.2], [4.8]]
Your answers are correct!
```

3.3.2. Gradient descent with regularization

To address overfitting, we can add a regularization term to control the values of parameters θ as shown in Equation (7):

$$\theta_j = \theta_j(1 - \alpha\frac{\lambda}{m}) - \alpha\frac{1}{m}\sum_{i=1}^{m}\left(h_\theta(x^{(i)}) - y^{(i)}\right)x_j^{(i)} \qquad \text{simultaneously update } \theta_j \text{ for}$$

Task 9:

1. compute the hypothesis value with your implemented function `hypothesis()` saved in "hyp" (1 line)
2. compute the difference between "hyp" and input y with function `np.substract()` , saved in "hyp_diff" (1 line)
3. compute the element-wise multiplication between "hyp_diff" and the j-th column of x with function `np.multiply()` , then saved into "error_list" (1 line)
4. compute the sum of errors with `np.sum()` , saved into `total_error` (1 line)
5. update each element of theta according to the equation (7). (1 line)

In [33]:

```
# Update theta with gradient descent and regularization term
def regularized_gradient_descent(theta, x, y, alpha, lamda):
    """
    Performs gradient descent with regulariztion to learn `theta`. Updates the
    i.e., one gradient step with learning rate `alpha`.

    Parameters
    ----------
    theta : array_like
        Initial values for the linear regression parameters.
        A vector of shape (n+1, 1).

    x : array_like
        The input dataset of shape (m, n+1).

    y : array_like
```

```
        Value at given features. A vector of shape (m, 1).

    alpha : float
        The learning rate.

    lamda : float
        hyperparameter for regularization term.

    Returns
    -------
    theta : array_like
        The learned linear regression parameters. A vector of shape (n+1, 1).

    cost : float
        J value in this iteration.

    Instructions
    ------------
    Peform a single gradient step on the parameter vector theta.
    """

    m = x.shape[0]
    n = theta.shape[0]
    new_theta = np.zeros((n, 1))

    # Task 9:
    # ====================== START YOUR CODE HERE ======================
    hyp = hypothesis(theta, x)
    hyp_diff = np.subtract(hyp, y)
    for j in range(n):
        x_column = np.reshape(x[:, j], (-1, 1)) # make sure this is a 2D arra
        error_list = np.multiply(hyp_diff, x_column)
        total_error = np.sum(error_list)
        new_theta[j] = np.subtract(np.multiply(theta[j],(1-np.divide(alpha*la
    # ====================== END YOUR CODE HERE ======================

    return new_theta
```

[Test Block 8]: Test code for function `regularized_gradient_descent()` . You can execute the code block, then it will print out whether you answer is correct or not.

In [34]:

```
# small demo for verification
demo_theta = np.array([1, 2, 3], ndmin=2).T # shape (3, 1)
demo_x = np.array([1, 2, 3], ndmin=2)
demo_y = np.array([20])
alpha = 0.1
lamda = 1

new_theta = regularized_gradient_descent(demo_theta, demo_x, demo_y, alpha, l
print('Updated theta value: [{}, {}, {}]'.format(new_theta[0], new_theta[1], 

if np.allclose(new_theta, np.array([[1.5], [3.0], [4.5]])):
    print('Your answers are correct!')
else:
    print('Your answers are not correct, please correct the funtion codes.')
```

```
Updated theta value: [[1.5], [3.], [4.5]]
Your answers are correct!
```

4. Optimization of Linear Regression Parameters

Your next task is to learn the parameters of linear regression with the given dataset with or without regularization terms.

4.1. Evaluation

You will implement the Mean Squared Error (MSE) function to evaluate the parameters.

$$\mathcal{E}(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \,. \tag{8}$$

Task 10: In this task, you will:

1. compute hypothesis (1 line)
2. compute the difference between "hyp" and input y with function `np.substract()` , saved in "hyp_diff" (1 line)
3. compute the squared errors list from "hyp_diff" and save it in "squared_errors" with `np.square()` (1 line)
4. compute mean of squared errors according to Equation (8) with `np.sum()` (1 line)

In [35]:

```
# computation of Mean Squared Error (MSE)
def evaluation(theta, x, y):
    """
        evaluates the sum of squares due to error.

        Parameters
        ----------
        theta : array_like
            Initial values for the linear regression parameters.
            A vector of shape (n+1, 1).

        x : array_like
            The input dataset of shape (m, n+1).

        y : array_like
            Value at given features. A vector of shape (m, 1).

        Returns
        -------
        sse : float
            the sum of squares due to error
    """

    mse = .0
    m = x.shape[0]
    # Task 10:
    # ====================== START YOUR CODE HERE ======================
    hyp = hypothesis(theta, x)
    hyp_diff = np.subtract(hyp, y)
    squared_errors = np.power(hyp_diff, 2)
    mse = np.divide(1,m,where=m!=0)*np.sum(squared_errors)
    # ====================== END YOUR CODE HERE ======================

    return mse
```

[Test Block 9]: Test code for function `evaluation()` .

In [36]:

```
# small demo for verification
```

```
demo_theta = np.array([1, 2, 3], ndmin=2).T # shape (3, 1)
demo_x = np.array([1, 2, 3], ndmin=2)
demo_y = np.array([16])

mse = evaluation(demo_theta, demo_x, demo_y)
print('Mean Squared Error: {}'.format(mse))

if mse == 4.0:
    print('Your answers are correct!')
else:
    print('Your answers are not correct, please correct the funtion codes.')
```

```
Mean Squared Error: 4.0
Your answers are correct!
```

4.2. Learning Parameters

You will use the following hyperparameters to run the linear regression learning algorithm.

In [37]:

```
# setting hyperparameters
alpha = 0.02 # learning rate
num_iters = 10000 # maximal iteration times
m, n = x_train.shape
```

4.2.1. Learning parameters without regularization

Task 11: You task are to:

1. compute current cost value (`cost_computation()`)
2. compute and update theta parameters with gradient descent (`gradient_descent()`)

In [38]:

```
# learned parameters
theta = np.random.rand(n, 1)
# record list
acc_train_list = list()
acc_test_list = list()
cost_list = list()
record_iters = list()
# training iterations
for k in range(num_iters):

    # Task 11:
    # ====================== START YOUR CODE HERE ======================
    # 1. compute current cost value
    # 2. compute and update theta parameters with gradient descent
    cost = cost_computation(theta, x_train, y_train)
    theta = gradient_descent(theta, x_train, y_train, alpha)
    # ======================= END YOUR CODE HERE =======================

    if k % 100 == 0:
        acc_train = .0
        acc_test = .0

        acc_train = evaluation(theta, x_train, y_train)
        acc_test = evaluation(theta, x_test, y_test)

        acc_train_list.append(acc_train)
        acc_test_list.append(acc_test)
        cost_list.append(cost)
        record_iters.append(k)
```

```
        # print output
        print('Iteration {}: training MSE: {:.4f}, testing MSE: {:.4f}'.forma
    if cost < 1.0e-4:
        break
```

```
Iteration 0: training MSE: 10.1187, testing MSE: 9.7848
Iteration 100: training MSE: 0.4668, testing MSE: 0.4387
Iteration 200: training MSE: 0.3888, testing MSE: 0.3689
Iteration 300: training MSE: 0.3490, testing MSE: 0.3346
Iteration 400: training MSE: 0.3210, testing MSE: 0.3105
Iteration 500: training MSE: 0.2994, testing MSE: 0.2920
Iteration 600: training MSE: 0.2821, testing MSE: 0.2771
Iteration 700: training MSE: 0.2679, testing MSE: 0.2648
Iteration 800: training MSE: 0.2560, testing MSE: 0.2545
Iteration 900: training MSE: 0.2458, testing MSE: 0.2457
Iteration 1000: training MSE: 0.2371, testing MSE: 0.2381
Iteration 1100: training MSE: 0.2295, testing MSE: 0.2314
Iteration 1200: training MSE: 0.2228, testing MSE: 0.2255
Iteration 1300: training MSE: 0.2168, testing MSE: 0.2202
Iteration 1400: training MSE: 0.2114, testing MSE: 0.2155
Iteration 1500: training MSE: 0.2066, testing MSE: 0.2112
Iteration 1600: training MSE: 0.2022, testing MSE: 0.2074
Iteration 1700: training MSE: 0.1983, testing MSE: 0.2038
Iteration 1800: training MSE: 0.1946, testing MSE: 0.2006
Iteration 1900: training MSE: 0.1913, testing MSE: 0.1976
Iteration 2000: training MSE: 0.1882, testing MSE: 0.1948
Iteration 2100: training MSE: 0.1853, testing MSE: 0.1923
Iteration 2200: training MSE: 0.1827, testing MSE: 0.1900
Iteration 2300: training MSE: 0.1802, testing MSE: 0.1878
Iteration 2400: training MSE: 0.1779, testing MSE: 0.1858
Iteration 2500: training MSE: 0.1758, testing MSE: 0.1839
Iteration 2600: training MSE: 0.1738, testing MSE: 0.1821
Iteration 2700: training MSE: 0.1720, testing MSE: 0.1805
Iteration 2800: training MSE: 0.1702, testing MSE: 0.1789
Iteration 2900: training MSE: 0.1686, testing MSE: 0.1775
Iteration 3000: training MSE: 0.1671, testing MSE: 0.1761
Iteration 3100: training MSE: 0.1656, testing MSE: 0.1749
Iteration 3200: training MSE: 0.1643, testing MSE: 0.1737
Iteration 3300: training MSE: 0.1630, testing MSE: 0.1726
Iteration 3400: training MSE: 0.1618, testing MSE: 0.1715
Iteration 3500: training MSE: 0.1607, testing MSE: 0.1705
Iteration 3600: training MSE: 0.1596, testing MSE: 0.1696
Iteration 3700: training MSE: 0.1586, testing MSE: 0.1687
Iteration 3800: training MSE: 0.1577, testing MSE: 0.1679
Iteration 3900: training MSE: 0.1568, testing MSE: 0.1671
Iteration 4000: training MSE: 0.1559, testing MSE: 0.1663
Iteration 4100: training MSE: 0.1551, testing MSE: 0.1656
Iteration 4200: training MSE: 0.1543, testing MSE: 0.1650
Iteration 4300: training MSE: 0.1536, testing MSE: 0.1643
Iteration 4400: training MSE: 0.1529, testing MSE: 0.1637
Iteration 4500: training MSE: 0.1522, testing MSE: 0.1631
Iteration 4600: training MSE: 0.1516, testing MSE: 0.1626
Iteration 4700: training MSE: 0.1510, testing MSE: 0.1621
Iteration 4800: training MSE: 0.1504, testing MSE: 0.1616
Iteration 4900: training MSE: 0.1499, testing MSE: 0.1611
Iteration 5000: training MSE: 0.1494, testing MSE: 0.1607
Iteration 5100: training MSE: 0.1489, testing MSE: 0.1602
Iteration 5200: training MSE: 0.1484, testing MSE: 0.1598
Iteration 5300: training MSE: 0.1479, testing MSE: 0.1594
Iteration 5400: training MSE: 0.1475, testing MSE: 0.1591
Iteration 5500: training MSE: 0.1471, testing MSE: 0.1587
Iteration 5600: training MSE: 0.1467, testing MSE: 0.1584
Iteration 5700: training MSE: 0.1463, testing MSE: 0.1581
Iteration 5800: training MSE: 0.1460, testing MSE: 0.1578
Iteration 5900: training MSE: 0.1456, testing MSE: 0.1575
Iteration 6000: training MSE: 0.1453, testing MSE: 0.1572
Iteration 6100: training MSE: 0.1450, testing MSE: 0.1569
Iteration 6200: training MSE: 0.1447, testing MSE: 0.1567
```

```
Iteration 6300: training MSE: 0.1444, testing MSE: 0.1564
Iteration 6400: training MSE: 0.1441, testing MSE: 0.1562
Iteration 6500: training MSE: 0.1438, testing MSE: 0.1560
Iteration 6600: training MSE: 0.1436, testing MSE: 0.1557
Iteration 6700: training MSE: 0.1433, testing MSE: 0.1555
Iteration 6800: training MSE: 0.1431, testing MSE: 0.1553
Iteration 6900: training MSE: 0.1428, testing MSE: 0.1551
Iteration 7000: training MSE: 0.1426, testing MSE: 0.1550
Iteration 7100: training MSE: 0.1424, testing MSE: 0.1548
Iteration 7200: training MSE: 0.1422, testing MSE: 0.1546
Iteration 7300: training MSE: 0.1420, testing MSE: 0.1545
Iteration 7400: training MSE: 0.1418, testing MSE: 0.1543
Iteration 7500: training MSE: 0.1416, testing MSE: 0.1542
Iteration 7600: training MSE: 0.1414, testing MSE: 0.1540
Iteration 7700: training MSE: 0.1413, testing MSE: 0.1539
Iteration 7800: training MSE: 0.1411, testing MSE: 0.1537
Iteration 7900: training MSE: 0.1410, testing MSE: 0.1536
Iteration 8000: training MSE: 0.1408, testing MSE: 0.1535
Iteration 8100: training MSE: 0.1407, testing MSE: 0.1534
Iteration 8200: training MSE: 0.1405, testing MSE: 0.1533
Iteration 8300: training MSE: 0.1404, testing MSE: 0.1532
Iteration 8400: training MSE: 0.1402, testing MSE: 0.1531
Iteration 8500: training MSE: 0.1401, testing MSE: 0.1530
Iteration 8600: training MSE: 0.1400, testing MSE: 0.1529
Iteration 8700: training MSE: 0.1399, testing MSE: 0.1528
Iteration 8800: training MSE: 0.1397, testing MSE: 0.1527
Iteration 8900: training MSE: 0.1396, testing MSE: 0.1526
Iteration 9000: training MSE: 0.1395, testing MSE: 0.1525
Iteration 9100: training MSE: 0.1394, testing MSE: 0.1524
Iteration 9200: training MSE: 0.1393, testing MSE: 0.1523
Iteration 9300: training MSE: 0.1392, testing MSE: 0.1523
Iteration 9400: training MSE: 0.1391, testing MSE: 0.1522
Iteration 9500: training MSE: 0.1390, testing MSE: 0.1521
Iteration 9600: training MSE: 0.1389, testing MSE: 0.1521
Iteration 9700: training MSE: 0.1388, testing MSE: 0.1520
Iteration 9800: training MSE: 0.1388, testing MSE: 0.1519
Iteration 9900: training MSE: 0.1387, testing MSE: 0.1519
```

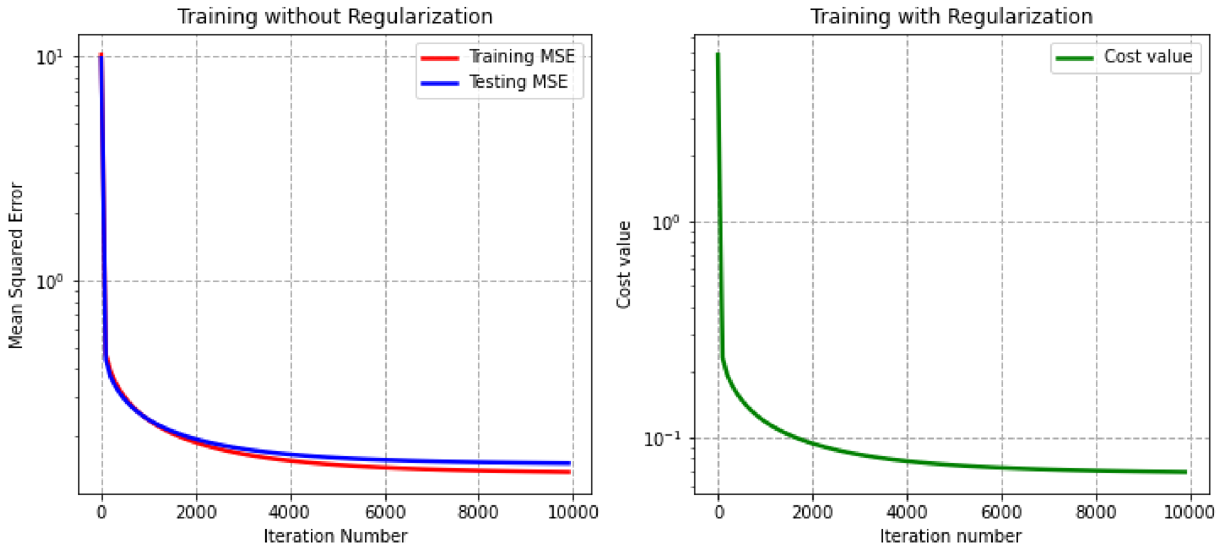
Visualization of learning process based on mean squared errors

In [39]:

```python
# training and testing accuracy visualization
fig = plt.figure(figsize=(12, 5))
(ax1, ax2) = fig.subplots(1, 2)

# figure 1 wiht axis 1
ax1.plot(record_iters, acc_train_list, color='r', linewidth=2.5, label='Train
ax1.plot(record_iters, acc_test_list, color='b', linewidth=2.5, label='Testin
ax1.set_yscale('log')
ax1.grid(linestyle='--', linewidth=1)
ax1.legend()
ax1.set_title('Training without Regularization')
ax1.set_xlabel('Iteration Number')
ax1.set_ylabel('Mean Squared Error');

# figure 2 wiht axis 2
ax2.plot(record_iters, cost_list, color='g', linewidth=2.5, label='Cost value
ax2.set_yscale('log')
ax2.grid(linestyle='--', linewidth=1)
ax2.legend(loc='upper right')
ax2.set_title('Training with Regularization')
ax2.set_xlabel('Iteration number')
ax2.set_ylabel('Cost value');
```



4.2.2. Learning parameters with regularization

Task 12: You task in this part is to:

1. compute current cost value (`regularized_cost_computation()`)
2. compute and update theta parameters with regularized gradient descent (`regularized_gradient_descent()`)

In [40]:

```
# learned parameters
regularized_theta = np.random.rand(n, 1)
lamda = 20
# record list
acc_train_list = list()
acc_test_list = list()
cost_list = list()
record_iters = list()

for k in range(num_iters):

    # Task 12:
    # ====================== START YOUR CODE HERE ======================
    # 1. compute current cost value
    # 2. compute and update theta parameters with regularized gradient descen
    cost = regularized_cost_computation(regularized_theta, x_train, y_train, 
    regularized_theta = regularized_gradient_descent(regularized_theta, x_tra
    # ====================== END YOUR CODE HERE ======================

    if k % 100 == 0:
        acc_train = .0
        acc_test = .0

        acc_train = evaluation(regularized_theta, x_train, y_train)
        acc_test = evaluation(regularized_theta, x_test, y_test)

        acc_train_list.append(acc_train)
        acc_test_list.append(acc_test)
        cost_list.append(cost)
        record_iters.append(k)

        # print output
        print('Iteration {}: training MSE: {:.4f}, testing MSE: {:.4f}'.forma
    if cost < 1.0e-4:
        break
```

```
Iteration 0: training MSE: 3.6255, testing MSE: 3.5633
Iteration 100: training MSE: 0.4887, testing MSE: 0.4727
Iteration 200: training MSE: 0.4055, testing MSE: 0.3938
Iteration 300: training MSE: 0.3619, testing MSE: 0.3532
Iteration 400: training MSE: 0.3336, testing MSE: 0.3269
Iteration 500: training MSE: 0.3128, testing MSE: 0.3076
Iteration 600: training MSE: 0.2963, testing MSE: 0.2923
Iteration 700: training MSE: 0.2828, testing MSE: 0.2798
Iteration 800: training MSE: 0.2714, testing MSE: 0.2692
Iteration 900: training MSE: 0.2615, testing MSE: 0.2601
Iteration 1000: training MSE: 0.2530, testing MSE: 0.2522
Iteration 1100: training MSE: 0.2455, testing MSE: 0.2451
Iteration 1200: training MSE: 0.2388, testing MSE: 0.2389
Iteration 1300: training MSE: 0.2328, testing MSE: 0.2333
Iteration 1400: training MSE: 0.2274, testing MSE: 0.2283
Iteration 1500: training MSE: 0.2225, testing MSE: 0.2237
Iteration 1600: training MSE: 0.2181, testing MSE: 0.2196
Iteration 1700: training MSE: 0.2140, testing MSE: 0.2158
Iteration 1800: training MSE: 0.2103, testing MSE: 0.2124
Iteration 1900: training MSE: 0.2069, testing MSE: 0.2092
Iteration 2000: training MSE: 0.2037, testing MSE: 0.2063
Iteration 2100: training MSE: 0.2008, testing MSE: 0.2036
Iteration 2200: training MSE: 0.1981, testing MSE: 0.2011
Iteration 2300: training MSE: 0.1956, testing MSE: 0.1988
Iteration 2400: training MSE: 0.1932, testing MSE: 0.1966
Iteration 2500: training MSE: 0.1910, testing MSE: 0.1946
Iteration 2600: training MSE: 0.1889, testing MSE: 0.1927
Iteration 2700: training MSE: 0.1870, testing MSE: 0.1910
Iteration 2800: training MSE: 0.1852, testing MSE: 0.1893
Iteration 2900: training MSE: 0.1835, testing MSE: 0.1878
Iteration 3000: training MSE: 0.1818, testing MSE: 0.1864
Iteration 3100: training MSE: 0.1803, testing MSE: 0.1850
Iteration 3200: training MSE: 0.1789, testing MSE: 0.1838
Iteration 3300: training MSE: 0.1775, testing MSE: 0.1826
Iteration 3400: training MSE: 0.1762, testing MSE: 0.1814
Iteration 3500: training MSE: 0.1750, testing MSE: 0.1804
Iteration 3600: training MSE: 0.1739, testing MSE: 0.1794
Iteration 3700: training MSE: 0.1728, testing MSE: 0.1784
Iteration 3800: training MSE: 0.1717, testing MSE: 0.1775
Iteration 3900: training MSE: 0.1707, testing MSE: 0.1767
Iteration 4000: training MSE: 0.1698, testing MSE: 0.1759
Iteration 4100: training MSE: 0.1689, testing MSE: 0.1751
Iteration 4200: training MSE: 0.1680, testing MSE: 0.1744
Iteration 4300: training MSE: 0.1672, testing MSE: 0.1737
Iteration 4400: training MSE: 0.1664, testing MSE: 0.1730
Iteration 4500: training MSE: 0.1656, testing MSE: 0.1724
Iteration 4600: training MSE: 0.1649, testing MSE: 0.1718
Iteration 4700: training MSE: 0.1642, testing MSE: 0.1712
Iteration 4800: training MSE: 0.1636, testing MSE: 0.1707
Iteration 4900: training MSE: 0.1630, testing MSE: 0.1702
Iteration 5000: training MSE: 0.1624, testing MSE: 0.1697
Iteration 5100: training MSE: 0.1618, testing MSE: 0.1692
Iteration 5200: training MSE: 0.1612, testing MSE: 0.1687
Iteration 5300: training MSE: 0.1607, testing MSE: 0.1683
Iteration 5400: training MSE: 0.1602, testing MSE: 0.1679
Iteration 5500: training MSE: 0.1597, testing MSE: 0.1675
Iteration 5600: training MSE: 0.1592, testing MSE: 0.1671
Iteration 5700: training MSE: 0.1588, testing MSE: 0.1668
Iteration 5800: training MSE: 0.1583, testing MSE: 0.1664
Iteration 5900: training MSE: 0.1579, testing MSE: 0.1661
Iteration 6000: training MSE: 0.1575, testing MSE: 0.1658
Iteration 6100: training MSE: 0.1571, testing MSE: 0.1655
Iteration 6200: training MSE: 0.1568, testing MSE: 0.1652
Iteration 6300: training MSE: 0.1564, testing MSE: 0.1649
Iteration 6400: training MSE: 0.1561, testing MSE: 0.1646
Iteration 6500: training MSE: 0.1557, testing MSE: 0.1644
Iteration 6600: training MSE: 0.1554, testing MSE: 0.1641
Iteration 6700: training MSE: 0.1551, testing MSE: 0.1639
Iteration 6800: training MSE: 0.1548, testing MSE: 0.1637
```

```
Iteration 6900: training MSE: 0.1545, testing MSE: 0.1634
Iteration 7000: training MSE: 0.1542, testing MSE: 0.1632
Iteration 7100: training MSE: 0.1539, testing MSE: 0.1630
Iteration 7200: training MSE: 0.1537, testing MSE: 0.1628
Iteration 7300: training MSE: 0.1534, testing MSE: 0.1626
Iteration 7400: training MSE: 0.1532, testing MSE: 0.1625
Iteration 7500: training MSE: 0.1529, testing MSE: 0.1623
Iteration 7600: training MSE: 0.1527, testing MSE: 0.1621
Iteration 7700: training MSE: 0.1525, testing MSE: 0.1620
Iteration 7800: training MSE: 0.1523, testing MSE: 0.1618
Iteration 7900: training MSE: 0.1521, testing MSE: 0.1617
Iteration 8000: training MSE: 0.1519, testing MSE: 0.1615
Iteration 8100: training MSE: 0.1517, testing MSE: 0.1614
Iteration 8200: training MSE: 0.1515, testing MSE: 0.1612
Iteration 8300: training MSE: 0.1513, testing MSE: 0.1611
Iteration 8400: training MSE: 0.1511, testing MSE: 0.1610
Iteration 8500: training MSE: 0.1510, testing MSE: 0.1609
Iteration 8600: training MSE: 0.1508, testing MSE: 0.1608
Iteration 8700: training MSE: 0.1506, testing MSE: 0.1606
Iteration 8800: training MSE: 0.1505, testing MSE: 0.1605
Iteration 8900: training MSE: 0.1503, testing MSE: 0.1604
Iteration 9000: training MSE: 0.1502, testing MSE: 0.1603
Iteration 9100: training MSE: 0.1500, testing MSE: 0.1602
Iteration 9200: training MSE: 0.1499, testing MSE: 0.1601
Iteration 9300: training MSE: 0.1498, testing MSE: 0.1601
Iteration 9400: training MSE: 0.1496, testing MSE: 0.1600
Iteration 9500: training MSE: 0.1495, testing MSE: 0.1599
Iteration 9600: training MSE: 0.1494, testing MSE: 0.1598
Iteration 9700: training MSE: 0.1493, testing MSE: 0.1597
Iteration 9800: training MSE: 0.1491, testing MSE: 0.1596
Iteration 9900: training MSE: 0.1490, testing MSE: 0.1596
```

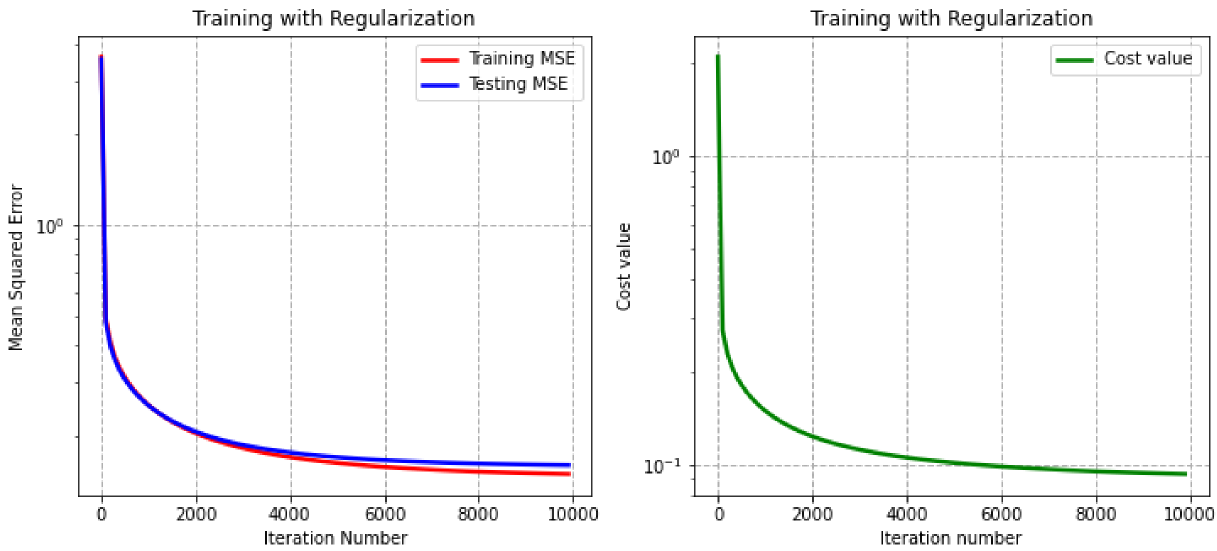
Visualization of learning process based on mean squared errors

In [41]:

```
# training and testing accuracy visualization
fig = plt.figure(figsize=(12, 5))
(ax1, ax2) = fig.subplots(1, 2)

ax1.plot(record_iters, acc_train_list, color='r', linewidth=2.5, label='Train
ax1.plot(record_iters, acc_test_list, color='b', linewidth=2.5, label='Testin
ax1.set_yscale('log')
ax1.grid(linestyle='--', linewidth=1)
ax1.legend()
ax1.set_title('Training with Regularization')
ax1.set_xlabel('Iteration Number')
ax1.set_ylabel('Mean Squared Error');

# figure 2 wiht axis 2
ax2.plot(record_iters, cost_list, color='g', linewidth=2.5, label='Cost value
ax2.set_yscale('log')
ax2.grid(linestyle='--', linewidth=1)
ax2.legend(loc='upper right')
ax2.set_title('Training with Regularization')
ax2.set_xlabel('Iteration number')
ax2.set_ylabel('Cost value');
```



5. Prediction of Sampled Data

To show the prediction effect of the learned parameters more intuitively, several sampled data items in testing dataset are used to predict the price of the used car. The data properties, real prices and the predicted prices are listed in the following table.

In [49]:

```
# ramdom data item generation from testing dataset
random_idx = np.random.randint(0, x_test.shape[0], size=5)
sample_x_test = x_test[random_idx, :]
sample_y_test = y_test[random_idx, :]

# sampled data visualization
raw_sample_data = pd.DataFrame(inverse_scale_feature(sample_x_test[:, 1:], sc
raw_sample_data['Real_Price'] = np.exp(sample_y_test)

pred_y_test_01 = np.exp(hypothesis(theta, sample_x_test))
pred_y_test_02 = np.exp(hypothesis(regularized_theta, sample_x_test))
raw_sample_data['Predicted_Price_01'] = pred_y_test_01 # without regularizati
raw_sample_data['Predicted_Price_02'] = pred_y_test_02 # with regularization
raw_sample_data.head()
```

Out[49]:

	Kilometers_Driven	Seats	Fuel_Consumption(kmpl)	Engine(CC)	Power(bhp)	Location_Ahm
0	51500.0	5.0	16.20	1599.0	103.20	
1	10200.0	5.0	19.10	1197.0	82.00	
2	71000.0	5.0	25.10	1498.0	98.60	
3	78000.0	7.0	11.40	2953.0	153.86	
4	47198.0	7.0	12.55	2982.0	168.50	

5 rows × 50 columns

6. Marking Scheme and Submission

This part carries 80% of the assignment grade. The Quiz posted on Moodle carries 20%. The marking scheme of this part follows.

Task	Mark
1. Min-max Scaling (`min_max_scaler()`)	4
2. Z-score Scaling (`z_score_scaler()`)	4
3. Dataset Split (`train_test_split()`)	4
4. Data Processing	2
5. Hypothesis (`hypothesis()`)	4
6. Cost Function (`cost_computation()`)	8
7. Regularized Cost Function (`cost_computation()`)	10
8. Gradient Descent (`gradient_descent()`)	10
9. Regularized Gradient Descent (`regularized_gradient_descent()`)	12
10. Evaluation (`evaluation()`)	10
11. Learning without regularization	6
12. Learning with regularization	6
TOTAL	**80**

Submission

You are required to upload to Moodle a zip file containing the following files.

1. Your completed Jupyter Notebook of this part. Please rename your file as `LinRegr_[SID]_[FirstnameLastname].ipynb` (where [SID] is your student ID and [FirstnameLastname] is your first name and last name concatenated) and do not include the data file. You must complete the **Acknowledgment** section in order for the file to be graded.
2. The PDF version (.pdf file) of your completed notebook (click `File > Download as > PDF via HTML` (If error occurs, you may download it as HTML and then save the HTML as PDF separately)).

In addition, please complete **A1Q: Assignment 1 -- Quiz** separately on the Moodle site.

7. Summary

Congratulations! You have implemented your first machine learning algorithm in this course! To summarize, you have prepared the data (by scaling and splitting them) for input to the linear regression (LR) learning algorithm, and implemented the hypothesis, cost function, regularization, and gradient descent optimization. You have run the algorithm to identify the optimal LR model using the training dataset, evaluated the performance of the model using the testing dataset, and applied the model to predicting prices of sampled data.