In [17]:

```
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
```

In [18]:

```
# Data loaded
bostan = load_boston()
```

In [19]:

```
# Data shape
bostan.data.shape
```

Out[19]:

```
(506, 13)
```

In [20]:

```
# Feature name
bostan.feature_names
```

Out[20]:

```
array(['CRIM', 'ZN', 'INDUS', 'CHAS', 'NOX', 'RM', 'AGE', 'DIS', 'RAD',
       'TAX', 'PTRATIO', 'B', 'LSTAT'], dtype='<U7')
```

In [21]:

```
# This is y value i.e. target
bostan.target.shape
```

Out[21]:

```
(506,)
```

In [22]:

```
# Convert it into pandas dataframe
data = pd.DataFrame(bostan.data, columns = bostan.feature_names)
data.head()
```

Out[22]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

In [23]:

```
# Statistical summary
data.describe()
```

Out[23]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.00000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.6740

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.29486
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.32000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.3775
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.4400
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.2250
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.9000

In [24]:

```
#noramlization for fast convergence to minima
data = (data - data.mean())/data.std()
data.head()
```

Out[24]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	-0.419367	0.284548	-1.286636	-0.272329	-0.144075	0.413263	-0.119895	0.140075	-0.981871	-0.665949	-1.457558	0.440616	-1.074499
1	-0.416927	-0.487240	-0.592794	-0.272329	-0.739530	0.194082	0.366803	0.556609	-0.867024	-0.986353	-0.302794	0.440616	-0.491953
2	-0.416929	-0.487240	-0.592794	-0.272329	-0.739530	1.281446	-0.265549	0.556609	-0.867024	-0.986353	-0.302794	0.396035	-1.207532
3	-0.416338	-0.487240	-1.305586	-0.272329	-0.834458	1.015298	-0.809088	1.076671	-0.752178	-1.105022	0.112920	0.415751	-1.360171
4	-0.412074	-0.487240	-1.305586	-0.272329	-0.834458	1.227362	-0.510674	1.076671	-0.752178	-1.105022	0.112920	0.440616	-1.025487

In [25]:

```
data.mean()
```

Out[25]:

```
CRIM       8.326673e-17
ZN         3.466704e-16
INDUS     -3.016965e-15
CHAS       3.999875e-16
NOX        3.563575e-15
RM        -1.149882e-14
AGE       -1.158274e-15
DIS        7.308603e-16
RAD       -1.068535e-15
TAX        6.534079e-16
PTRATIO   -1.084420e-14
B          8.117354e-15
LSTAT     -6.494585e-16
dtype: float64
```

In [26]:

```
#from sklearn.preprocessing import StandardScaler
#std = StandardScaler()
#data = std.fit_transform(data)
#data
# MEDV(median value is usually target), change it to price
data["PRICE"] = bostan.target
data.head()
```

Out[26]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE
0	-0.419367	0.284548	-1.286636	-0.272329	-0.144075	0.413263	-0.119895	0.140075	-0.981871	-0.665949	-1.457558	0.440616	-1.074499	24.0
1	-0.416927	-0.487240	-0.592794	-0.272329	-0.739530	0.194082	0.366803	0.556609	-0.867024	-0.986353	-0.302794	0.440616	-0.491953	21.6
2	-0.416929	-0.487240	-0.592794	-0.272329	-0.739530	1.281446	-0.265549	0.556609	-0.867024	-0.986353	-0.302794	0.396035	-1.207532	34.7
3	-0.416338	-0.487240	-1.305586	-0.272329	-0.834458	1.015298	-0.809088	1.076671	-0.752178	-1.105022	0.112920	0.415751	-1.360171	33.4
4	-0.412074	-0.487240	-1.305586	-0.272329	-0.834458	1.227362	-0.510674	1.076671	-0.752178	-1.105022	0.112920	0.440616	-1.025487	36.2

In [27]:

```
# Target and features
Y = data["PRICE"]
X = data.drop("PRICE", axis = 1)
```

In [28]:

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size = 0.3)
print(x_train.shape, x_test.shape, y_train.shape, y_test.shape)
```

(354, 13) (152, 13) (354,) (152,)

In [29]:

```
#x_train = (x_train - x_train.mean())/ x_train.std()
#x_test = (x_test - x_train.mean())/ x_test.std()
#std = StandardScaler()
#x_train = std.fit_transform(x_train)
#x_test = std.fit_transform(x_test)
#x_train[0:,0:5]
x_train["PRICE"] = y_train
#x_test["PRICE"] = y_test
```

```
C:\Users\Lab2-3\Anaconda3\lib\site-packages\ipykernel_launcher.py:7: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy
  import sys
```

In [30]:

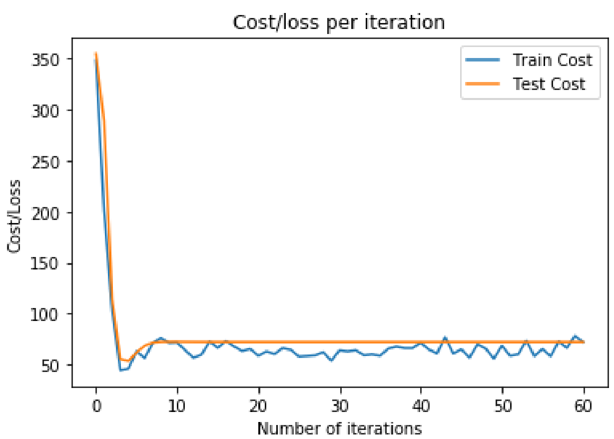
```
#x_train["PRICE"] = y_train
#x_test["PRICE"] = y_test
def cost_function(b, m, features, target):
    totalError = 0
    for i in range(0, len(features)):
        x = features
        y = target
        totalError += (y[:,i] - (np.dot(x[i] , m) + b)) ** 2
    return totalError / len(x)
```

In [31]:

```
# The total sum of squares (proportional to the variance of the data)i.e. ss_tot
# The sum of squares of residuals, also called the residual sum of squares i.e. ss_res
# the coefficient of determination i.e. r^2(r squared)
def r_sq_score(b, m, features, target):
    for i in range(0, len(features)):
        x = features
        y = target
        mean_y = np.mean(y)
        ss_tot = sum((y[:,i] - mean_y) ** 2)
        ss_res = sum(((y[:,i]) - (np.dot(x[i], m) + b)) ** 2)
        r2 = 1 - (ss_res / ss_tot)
    return r2
def gradient_decent(w0, b0, train_data, x_test, y_test, learning_rate):
    n_iter = 500
    partial_deriv_m = 0
    partial_deriv_b = 0
    cost_train = []
    cost_test = []
    for j in range(1, n_iter):

        # Train sample
        train_sample = train_data.sample(160)
        y = np.asmatrix(train_sample["PRICE"])
        x = np.asmatrix(train_sample.drop("PRICE", axis = 1))
        # Test sample
        #x_test["PRICE"] = [y_test]
        #test_data = x_test
        #test_sample = test_data.sample()
        #y_test = np.asmatrix(test_sample["PRICE"])
        #x_test = np.asmatrix(test_sample.drop("PRICE", axis = 1))

        for i in range(len(x)):
            partial_deriv_m += np.dot(-2*x[i].T , (y[:,i] - np.dot(x[i] , w0) + b0))
            partial_deriv_b += -2*(y[:,i] - (np.dot(x[i] , w0) + b0))

        w1 = w0 - learning_rate * partial_deriv_m
        b1 = b0 - learning_rate * partial_deriv_b
```

```python
        b1 = b0 - learning_rate * partial_deriv_b

        if (w0==w1).all():
            #print("W0 are\n", w0)
            #print("\nW1 are\n", w1)
            #print("\n X are\n", x)
            #print("\n y are\n", y)
            break
        else:
            w0 = w1
            b0 = b1
            learning_rate = learning_rate/2


        error_train = cost_function(b0, w0, x, y)
        cost_train.append(error_train)
        error_test = cost_function(b0, w0, np.asmatrix(x_test), np.asmatrix(y_test))
        cost_test.append(error_test)

        #print("After {0} iteration error = {1}".format(j, error_train))
        #print("After {0} iteration error = {1}".format(j, error_test))

    return w0, b0, cost_train, cost_test
# Run our model
learning_rate = 0.001
w0_random = np.random.rand(13)
w0 = np.asmatrix(w0_random).T
b0 = np.random.rand()

optimal_w, optimal_b, cost_train, cost_test = gradient_decent(w0, b0, x_train, x_test, y_test, learning_rate)
print("Coefficient: {} \n y_intercept: {}".format(optimal_w, optimal_b))

'''
error = cost_function(optimal_b, optimal_w, np.asmatrix(x_test), np.asmatrix(y_test))
print("Mean squared error:",error)
'''

plt.figure()
plt.plot(range(len(cost_train)), np.reshape(cost_train,[len(cost_train), 1]), label = "Train Cost")
plt.plot(range(len(cost_test)), np.reshape(cost_test, [len(cost_test), 1]), label = "Test Cost")
plt.title("Cost/loss per iteration")
plt.xlabel("Number of iterations")
plt.ylabel("Cost/Loss")
plt.legend()
plt.show()

#error = cost_function(optimal_b, optimal_w, np.asmatrix(x_test), np.asmatrix(y_test))
#print("Mean squared error: %.2f" % error)
```

```
Coefficient: [[-1.30009227]
 [-0.33305961]
 [ 2.04651708]
 [ 1.39860121]
 [ 1.88755068]
 [ 2.90880097]
 [ 1.32260796]
 [-1.721249  ]
 [-0.73217443]
 [-1.17508767]
 [-1.28841261]
 [-0.13753401]
 [-2.36580324]]
 y_intercept: [[21.48151536]]
```



In [ ]:

```

```