Lab1_LinearRegression

October 29, 2020

1 Lab 1 : Linear regression with NumPy

In this exercise, you will implement a linear regressor for the house pricing problem. You will predict house prices training your predictor on previous collected data.

The idea behind (multiple) linear regression is to extract a linear relation between features and target data which can be represented by the following equation:

$$y(x_1, x_2, \ldots, x_n) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n = w_0 + \sum_{i=1}^{m} w_i \cdot x_i. \tag{1}$$ (1)

[1]:
```
#####################################################
# Imports
#####################################################

import numpy as np
import math
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from mpl_toolkits import mplot3d


# Settings
plt.style.use('seaborn-white')


#####################################################
# Check function tool, skip this
#####################################################

def check(func, print_log=False):

    # Set seed
    np.random.seed(1234)

    # Select function to check
```

```
    if func.__name__ == 'predict':
        args = {
            'x': np.random.normal(0, 1, size=(10, 501)),
            'w': np.random.normal(0, 1, size=(501, 1)),
        }
        res = 84.37925870442523
        res_ = func(**args).sum()
        cond = (res_.sum() - res) < 1e-8

    elif func.__name__ == 'compute_cost':
        args = {
            'x': np.random.normal(0, 1, size=(10, 501)),
            'y': np.random.normal(0, 1, size=(10, 1)),
            'w': np.random.normal(0, 1, size=(501, 1)),
        }
        res = 131.13466658728424
        res_ = func(**args)
        cond = (res_ - res) < 1e-8

    elif func.__name__ == 'compute_cost_multivariate':
        args = {
            'x': np.random.normal(0, 1, size=(10, 501)),
            'y': np.random.normal(0, 1, size=(10, 1)),
            'w': np.random.normal(0, 1, size=(501, 1)),
        }
        res = 131.13466658728424
        res_ = func(**args)
        cond = (res_ - res) < 1e-8

    elif func.__name__ == 'gradient_descent':
        args = {
            'x': np.random.normal(0, 1, size=(10, 501)),
            'y': np.random.normal(0, 1, size=(10, 1)),
            'w': np.random.normal(0, 1, size=(501, 1)),
            'learning_rate': 0.005,
            'num_iters': 10,
        }
        res = [
            158.544797156765,
            2.203001889427611,
            25.109538060963843,
        ] # Sums of the arryays
        res_ = func(**args)
        cond = all([(r_.sum() - r) < 1e-8 for r, r_ in zip(res, res_)])

    else:
```

```
        raise Exception(f'Error. The check of the function {func.__name__} is␣
↪not implemented.')

    if cond:
        print(f'Your function "{func.__name__}" is correct!')
    else:
        print(f'Your function "{func.__name__}" is NOT correct!')

    # Print output log
    if print_log:
        if isinstance(res, list):
            for r, r_ in zip(res, res_):
                if isinstance(r_, np.ndarray):
                    r_ = r_.sum()
                print(f'Your output: {r_}, expected output: {r}')
        if isinstance(res, float) or isinstance(res, int) or isinstance(res,␣
↪str):
            print(f'Your output: {res_}, expected output: {res}')
```

Dataset description

The Boston Housing dataset contains prices of various houses in Boston depending on several parameters. The dataset contains 506 samples and 13 features. Your task is to fit a linear model and predict prices using unseen data.

[2]:
```
# Load dataset
dataset = load_boston()

# Print dataset info
print(dataset.data.shape)
print(dataset.keys())
print(dataset.feature_names)
print(dataset.DESCR)
```

```
(506, 13)
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename'])
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
.. _boston_dataset:

Boston house prices dataset
---------------------------

**Data Set Characteristics:**

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive. Median Value
```

```
(attribute 14) is usually the target.

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000
sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0
otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by
town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/


This dataset was taken from the StatLib library which is maintained at Carnegie
Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
…', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that
address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential
Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In
Proceedings on the Tenth International Conference of Machine Learning, 236-243,
```

University of Massachusetts, Amherst. Morgan Kaufmann.

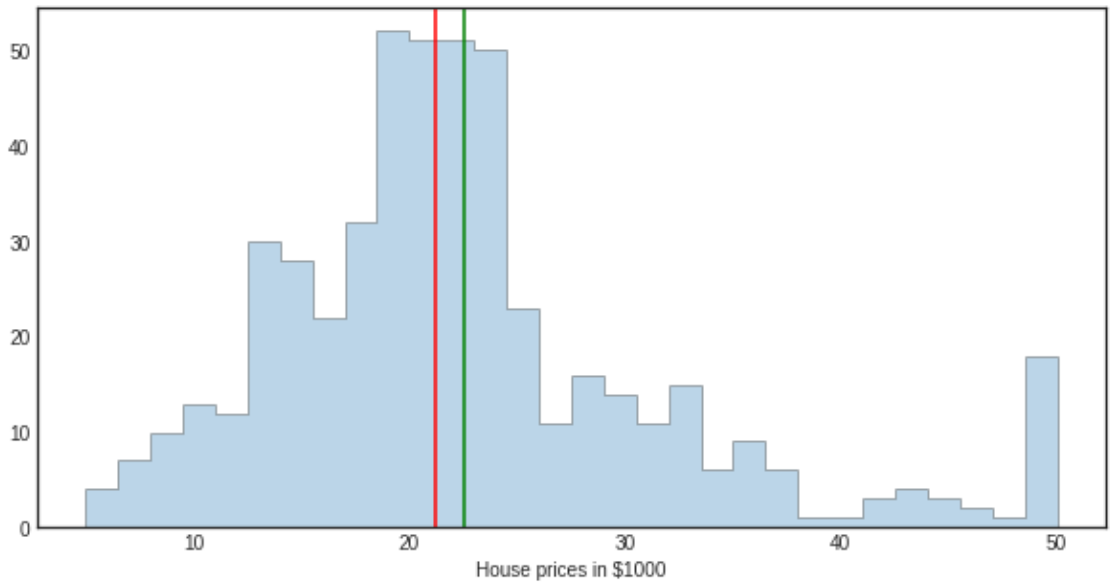
Data analysis

Before training a linear regressor, we will visualize the relationship between each feature and target data using histogram and scatter plots from the matplotlib library. Let's first plot the distribution of the target variable.

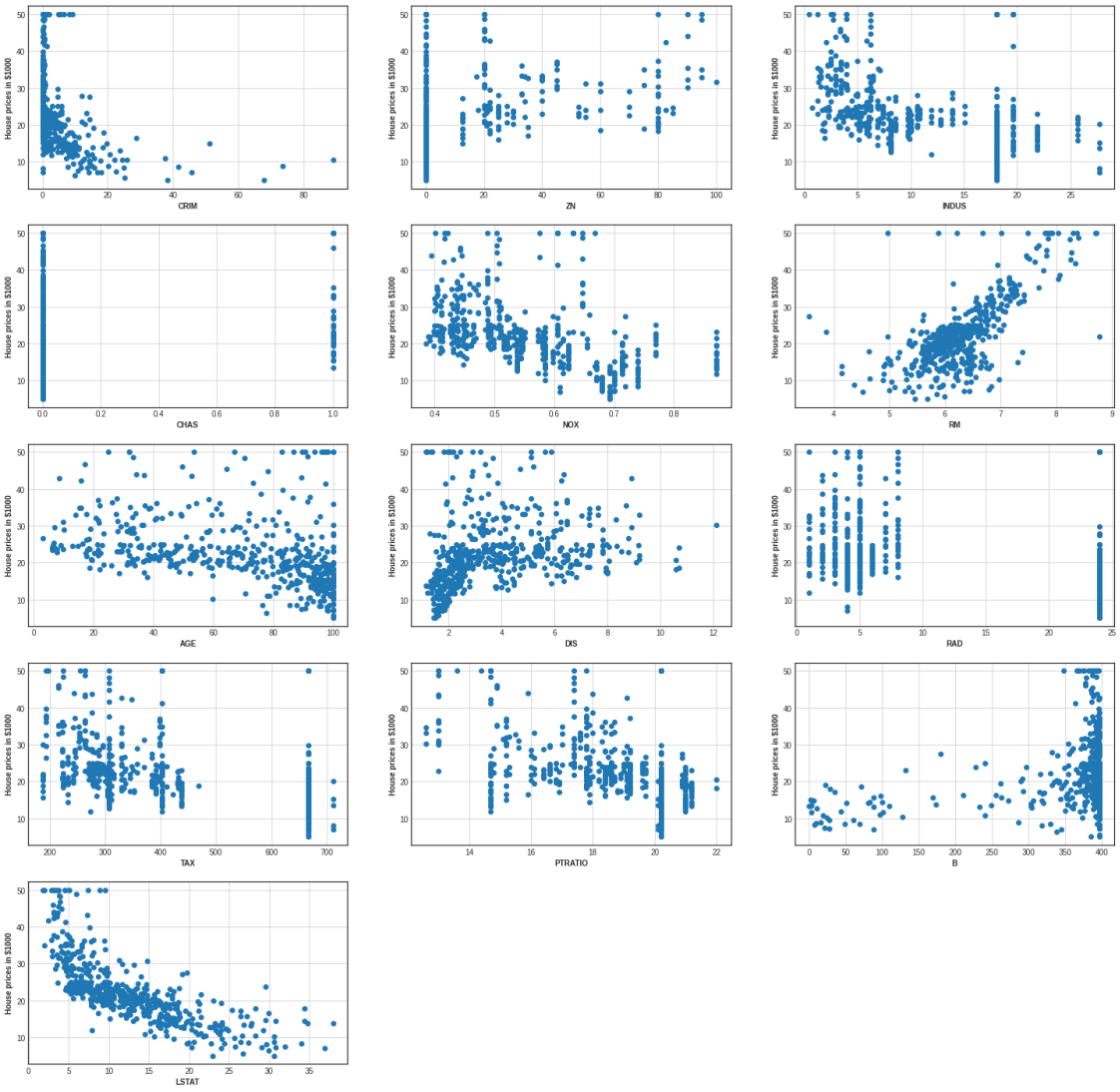
[3]:
```
# Retrieve input features and target prices
x = np.array(dataset.data) # Input data of shape [num_samples, num_feat]
y = np.array(dataset.target) # Targets of shape [num_samples]

# Info
print(f'Dataset shape -> {x.shape}, target variable shape -> {y.shape}')
```

```
Dataset shape -> (506, 13), target variable shape -> (506,)
```

[4]:
```
# Price distribution
plt.figure(figsize=(10, 5))
kwargs = dict(histtype='stepfilled', alpha=0.3, density=False, bins=30, ec="k")
plt.hist(y, **kwargs)
plt.axvline(x=np.mean(y), color='g')
plt.axvline(x=np.median(y), color='r')
plt.xlabel("House prices in $1000")
plt.show()
plt.show()
```



[5]:
```
# Plot (feature, target) plot, for each single feature
plt.figure(figsize=(25, 25))
for idx_f, feat_name in enumerate(dataset.feature_names):
  plt.subplot(5, 3 , idx_f + 1)
  plt.scatter(x[:, idx_f], y, marker='o')
  plt.xlabel(feat_name, fontweight='bold')
  plt.ylabel('House prices in $1000', fontweight='bold')
  plt.grid(True)
plt.show()
```



Linearity and correlation

From the above charts we can see that LSTAT and RM could follow some linearity in the data, we could even see come correlation between the variables.

Simple linear regression

In simple linear regression, a single independent variable is used to predict the value of a dependent variable. Since it is hard to *visualize* our model with multiple features, we will first predict the house prices with just one variable and then move to the regression with all the features. For example, we can select the 'LSTAT' feature which shows a negative correlation with the house prices.

[6]:
```
# Select LSTAT feature from X
x_lstat = x[:, -1].reshape(-1,1)
y_price = y.reshape(-1,1)

# Print info
print(x_lstat.shape)
print(y_price.shape)
```

```
(506, 1)
(506, 1)
```

[7]:
```
# Split dataset into training and testing sets
x_train, x_test, y_train, y_test = train_test_split(x_lstat, y_price,␣
 ↪test_size=0.2, random_state=5)
num_tr = len(x_train)
num_feat = x_train.shape[-1]

# Append intercept term to x_train
print(x_train.shape)
x_train = np.concatenate([x_train, np.ones([num_tr, 1])], -1)
print(x_train. shape)
print(f'Total samples in X_train: {num_tr}')
```

```
(404, 1)
(404, 2)
Total samples in X_train: 404
```

The objective of linear regression is to minimize the cost function

$$J(w) = \frac{1}{2m}\sum_{i=1}^{m}(h_w(x^{(i)}) - y^{(i)})^2 \qquad (2) \qquad (2)$$

where the hypothesis $h_w(x)$ is given by the linear model

$$h_w(x) = w^T x = w_0 + w_1 x_1. \qquad (3) \qquad (3)$$

In each iteration, we perform the following update:

$$w_j := w_j - \alpha\frac{1}{m}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad (4) \qquad (4)$$

2 Ex.1

Complete the following code to compute prediction as reported in Equation (3).

[8]:
```
print(x_train.shape)
print(x_train.shape)
```

```
(404, 2)
(404, 2)
```

[9]:
```
#####################################################
# Compute the prediction
#####################################################

def predict(x, w):
    """
    Compute the prediction of a linear model.
    Inputs:
        x: np.ndarray input data of shape [num_samples, num_feat + 1]
        w: np.ndarray weights of shape [num_feat + 1, 1]
    Outputs:
        h: np.ndarray predictions of shape [num_samples, 1]
    """
    res = np.dot(x, w)
    return  res

    ##################################
    pass

# Test your code -> uncomment
check(predict)
```

```
Your function "predict" is correct!
```

3 Ex.2

Complete the following code to compute the cost function as reported in Equation (2).

[10]:
```
#####################################################
# Loss function (mean squared error -> MSE)
#####################################################

def compute_cost(x, y, w):
    """
    Inputs:
        x: np.ndarray input data of shape [num_samples, num_feat + 1]
        y: np.ndarray targets data of shape [num_samples, 1]
        w: np.ndarray weights of shape [num_feat + 1, 1]
    Outputs:
        mse: scalar.
    """
```

```
    ##### WRITE YOUR CODE HERE #####
    hyp_val = predict(x, w)
    summ = np.sum(np.power((hyp_val - y), 2)) / (2*len(hyp_val))
    #print(summ)
    return summ
    ################################
    pass

# Test your code -> uncomment
check(compute_cost)
```

```
Your function "compute_cost" is correct!
```

4 Gradient descent algorithm

Repeat until convergence (or maximum number of iterations) { 1. Calculate gradient 2. Multiply by learning rate 3. Subtract from weights

}

5 Ex.3

Implement gradient descent algorithm as reported in Equation (4).

In each iteration, we perform the following update:

$$w_j := w_j - \alpha \frac{1}{m}(h_w(x^{(i)}) - y^{(i)})x_j^{(i)} \qquad (4) \qquad (5)$$

[11]:
```
def gradient_descent(x, y, w, learning_rate, num_iters):
    """
    Inputs:
        x: np.ndarray input data of shape [num_samples, num_feat + 1]
        y: np.ndarray targets data of shape [num_samples, 1]
        w: np.ndarray weights of shape [num_feat + 1, 1]
        learning_rate: scalar, the learning rate.
        num_iters: int, the number of iterations.
    Outputs:
        j_hist: list of loss values of shape [num_iters + 1]
        w_opt: [num_feat + 1, 1] which is the output of the gradient descent
        w_hist: [num_feat + 1, num_iters + 1]
    """
    ##### WRITE YOUR CODE HERE #####
    j_hist = []
    w_hist = []
```

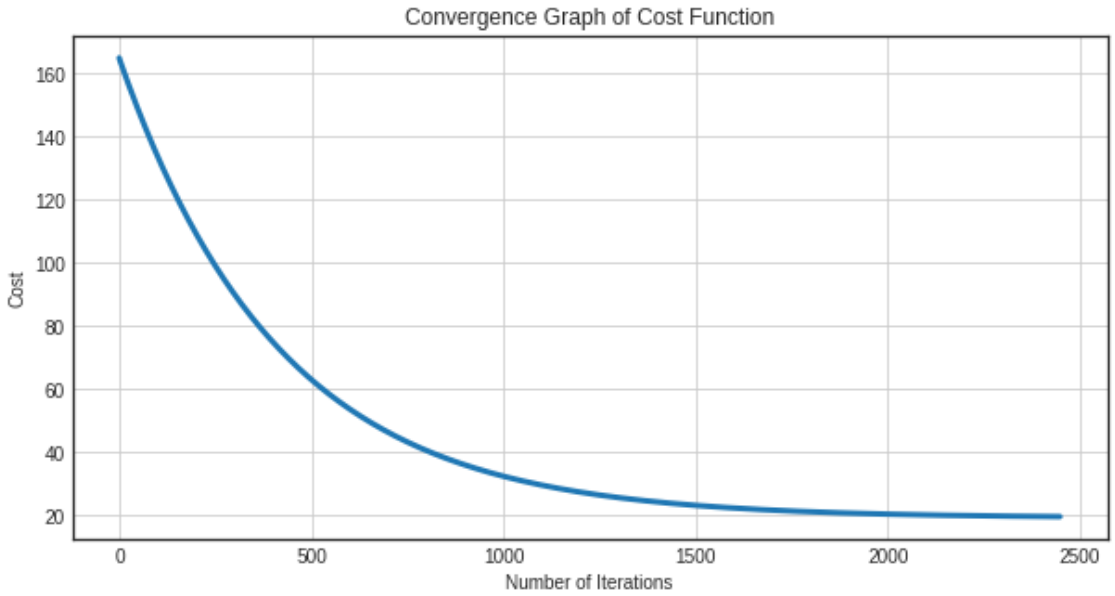
```
    #TODO: how do we check the convergence, by looking if the cost function is␣
 ↪getting lower during the iterations???
    for i in range(num_iters):
        y_predicted = predict(x, w)
        residual = (y_predicted - y)
        gradient = np.dot(x.T, residual)
        w_hist.append(w)
        w = w - learning_rate/len(y) * gradient
        j_hist.append(compute_cost(x, y, w))
        if(i > 3 and  (math.isclose(j_hist[i-3],j_hist[i],␣
 ↪rel_tol=1e-04,abs_tol=0.00))):
            break
    return(np.array(j_hist), w, np.array(w_hist))
    pass

# Test your code -> uncomment
check(gradient_descent)
```

```
Your function "gradient_descent" is correct!
```

[12]:
```
# Initialize the parameters of the linear model
w = np.zeros([num_feat + 1, 1])
#maybe better to use one instead of zeros, the initial cost is better with ones.
#w = np.zeros([num_feat + 1, 1])

# Parameters for the gradient descent
num_iters = 3000
learning_rate = 0.005

# Compute the initial cost
initial_cost = compute_cost(x_train, y_train, w)
print("Initial cost is: ", initial_cost)

# Apply gradient descent algorithm
j_hist, w_opt, w_hist = gradient_descent(x_train, y_train, w, learning_rate,␣
 ↪num_iters)
print("Optimal parameters are: \n", w_opt)
print("Iteration are", len(j_hist))
print("Final cost is: ", j_hist[-1])

# Plot loss history
plt.figure(figsize=(10, 5))
plt.plot(range(len(j_hist)), j_hist, lw=3)
plt.title("Convergence Graph of Cost Function")
plt.xlabel("Number of Iterations")
plt.ylabel("Cost")
plt.grid()
```

```
plt.show()
```

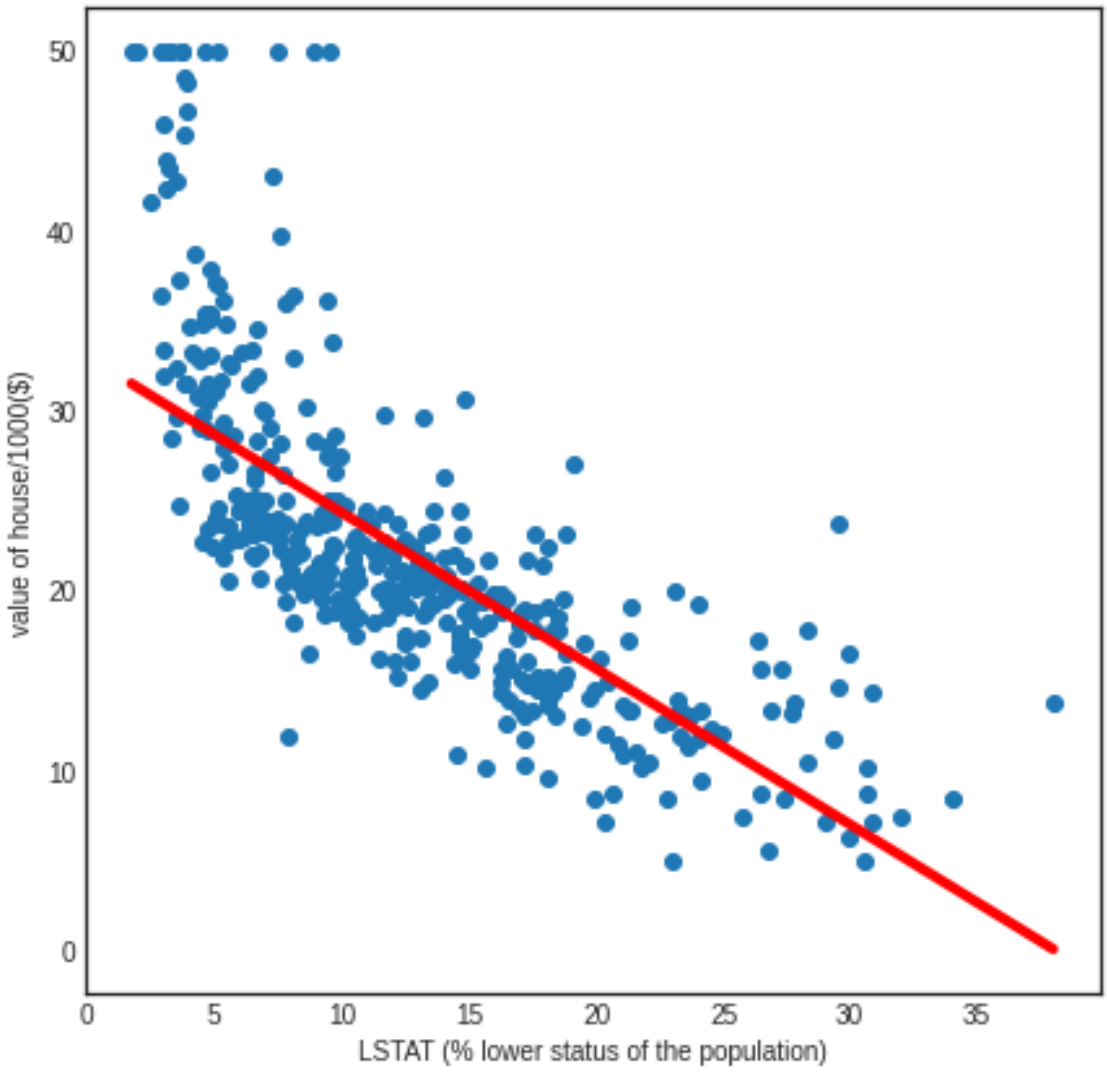
```
Initial cost is:  299.38922029702974
Optimal parameters are:
 [[-0.86786752]
 [33.03487396]]
Iteration are 2449
Final cost is:  19.63598351575988
```



Plotting the model fitted line on the output variable.

[13]:
```
# Plot the model fitted line on the output variable
plt.figure(figsize=(7, 7))
prediction_space = np.linspace(x_lstat.min(), x_lstat.max()).reshape(-1,1)
prediction_space = np.concatenate([prediction_space, np.
 ↪ones([len(prediction_space), 1])], -1)
plt.scatter(x_train[:, 0], y_train)
plt.plot(
    prediction_space[:, 0],
    predict(
        prediction_space,
        w_opt
    ),
    color='r', linewidth=4
)
plt.ylabel('value of house/1000($)')
plt.xlabel('LSTAT (% lower status of the population)')
```

```
plt.show()
```



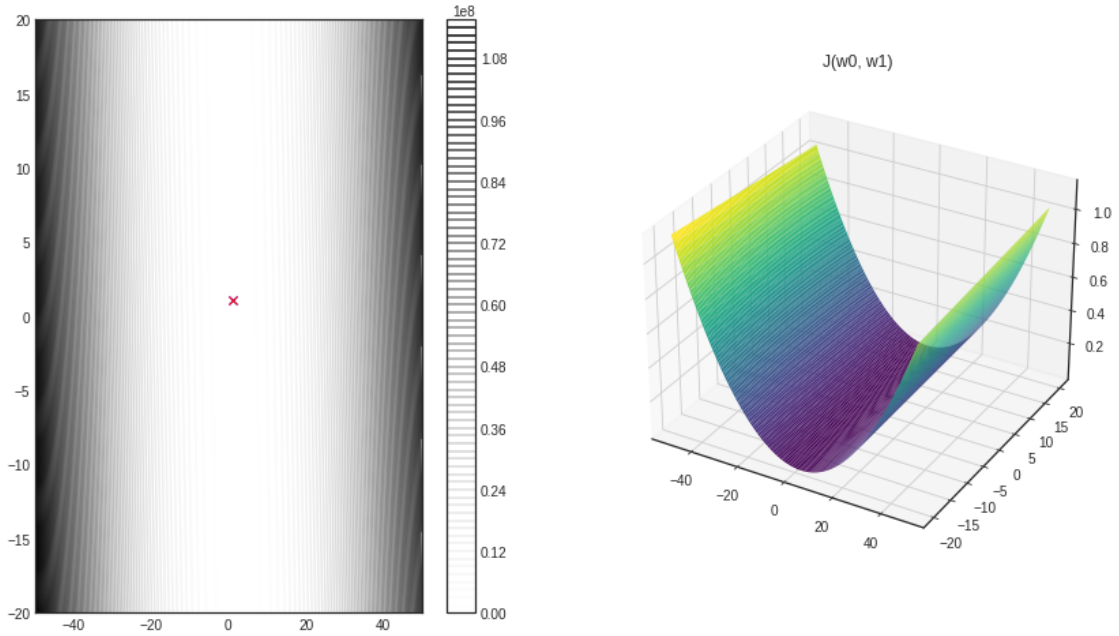
6 Visualizing $J(w)$ (Optional)

To better understand the cost function, you can plot the cost over a 2-dimensional grid of w_0 and w_1 values.

[35]:
```
fig = plt.figure(figsize=(14,8)) # create the canvas for plotting
ax1 = fig.add_subplot(1,2,1)

# Create grid of w0/w1 values
w0_values = np.linspace(-50, 50, 100);
w1_values = np.linspace(-20, 20, 100);
```

```
W0, W1 = np.meshgrid(w0_values, w1_values)
J = np.zeros((w0_values.shape[0] * w1_values.shape[0]))

# Compute cost function for each point in the grid
for i, (w0,w1) in enumerate(zip(np.ravel(W0), np.ravel(W1))):
  w = [w0, w1]
  J[i] = compute_cost(x_train, y_train, w)

J = J.reshape(w0_values.shape[0], w1_values.shape[0])

# Plot cost function
plt.contour(W0, W1, J, 100)
plt.colorbar()

# Plot params history
plt.plot(w_hist[0, 0], w_hist[1, 0], 'bx', # Initial position
         w_hist[0, -1], w_hist[1, -1], 'rx', # Final position
         w_hist[0, ::10], w_hist[1, ::10], 'r--')

ax2 = fig.add_subplot(122, projection='3d')
ax2.plot_surface(W0, W1, J, rstride=1, cstride=1,
                 cmap='viridis', edgecolor='none')
ax2.set_title('J(w0, w1)')
```

[35]: Text(0.5, 0.92, 'J(w0, w1)')



7 Multiple Linear Regression

We will now consider all the features of our dataset. Nevertheless, some features may differ by orders of magnitude. For this reason, we firstly perform feature scaling since it can make gradient descent converge much more quickly.

We will consider a normalization technique called Standardization which is based on mean and standard deviation of X as follows:

$$z = \frac{x - \mu}{\sigma} \tag{5}$$ (6)

where

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x_i, \quad \sigma = \sqrt{\frac{1}{m}\sum_{i=1}^{m}(x_i - \mu)^2}. \tag{6}$$ (7)

[14]:
```
# Split the dataset into train and test set
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2,␣
 ↪random_state=42)
num_tr = len(x_train)
num_feat = x_train.shape[-1]

# Normalize the data
mu = x_train.mean(axis=0) # Mean from the train set
sigma = x_train.std(axis=0) # STD from the train set
x_train = (x_train - mu) / sigma
x_test = (x_test - mu) / sigma # Normalize with stats from the train
y_train = y_train.reshape(-1, 1)

# Add intercept term
x_train = np.concatenate([x_train, np.ones([num_tr, 1])], -1)
x_test = np.concatenate([x_test, np.ones([len(x_test), 1])], -1)
print(x_train.shape)
```

```
(404, 14)
```

Implementation detail: For the multivariate case, we can consider the vectorized form of the cost function:

$$J(w) = \frac{1}{2m}(Xw - y)^T(Xw - y) \tag{7}$$ (8)

8 Ex.4

Implement the cost function for the multivariate case as reported in Equation (7).

[15]:
```
##################################################
# Loss function (mean squared error -> MSE)
##################################################

def compute_cost_multivariate(x, y, w):
```

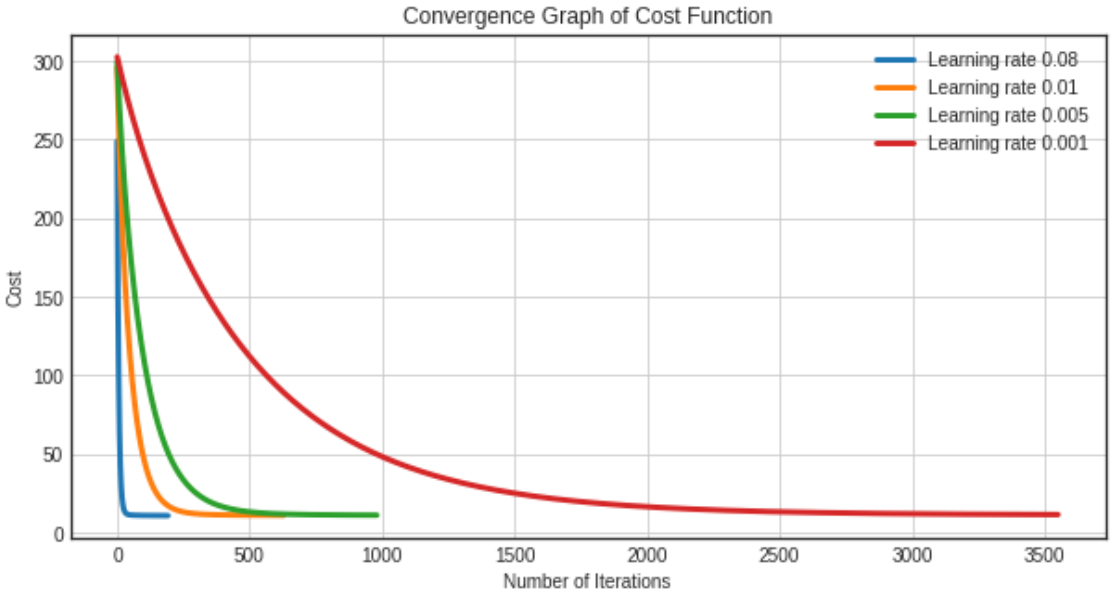
```
    """
    Inputs:
        x: np.ndarray input data of shape [num_samples, num_feat + 1]
        y: np.ndarray targets data of shape [num_samples, 1]
        w: np.ndarray weights of shape [num_feat + 1, 1]
    Outputs:
        mse: scalar.
    """

    ##### WRITE YOUR CODE HERE #####
    error = (np.dot(x, w) - y)
    return (np.dot(error.T, error)) / (2*len(y))
    ################################
    pass

# Test your code -> uncomment
check(compute_cost_multivariate, print_log=True)
```

```
Your function "compute_cost_multivariate" is correct!
Your output: [[131.13466659]], expected output: 131.13466658728424
```

[23]:
```
# Initialize the weights of the linear model
w = np.zeros((num_feat + 1, 1))

# Parameters of the gradient descent
num_iters = 5000
learning_rate = [ 0.08 ,0.01, 0.005, 0.001]

# Comput the initial cost
initial_cost = compute_cost_multivariate(x_train, y_train, w)
print("Initial cost is: ", initial_cost, "\n")


plt.figure(figsize=(10, 5))
#plt.plot(range(len(j_hist)), j_hist, lw=3)
plt.title("Convergence Graph of Cost Function")
plt.xlabel("Number of Iterations")
plt.ylabel("Cost")
plt.grid()
# Gradient descent
plots = []
for rate in learning_rate:
    (j_hist, w_opt, _) = gradient_descent(x_train, y_train, w, rate, num_iters)
    line, = plt.plot(range(len(j_hist)), j_hist, lw=3, label= ('Learning rate '␣
 ↪+ str(rate)))
    plots.append(line)
# Output info
```

```
print("Optimal parameters are: \n", w_opt, "\n")
print("Final cost is: ", j_hist[-1])
plt.legend(handles=plots)
plt.show()
```

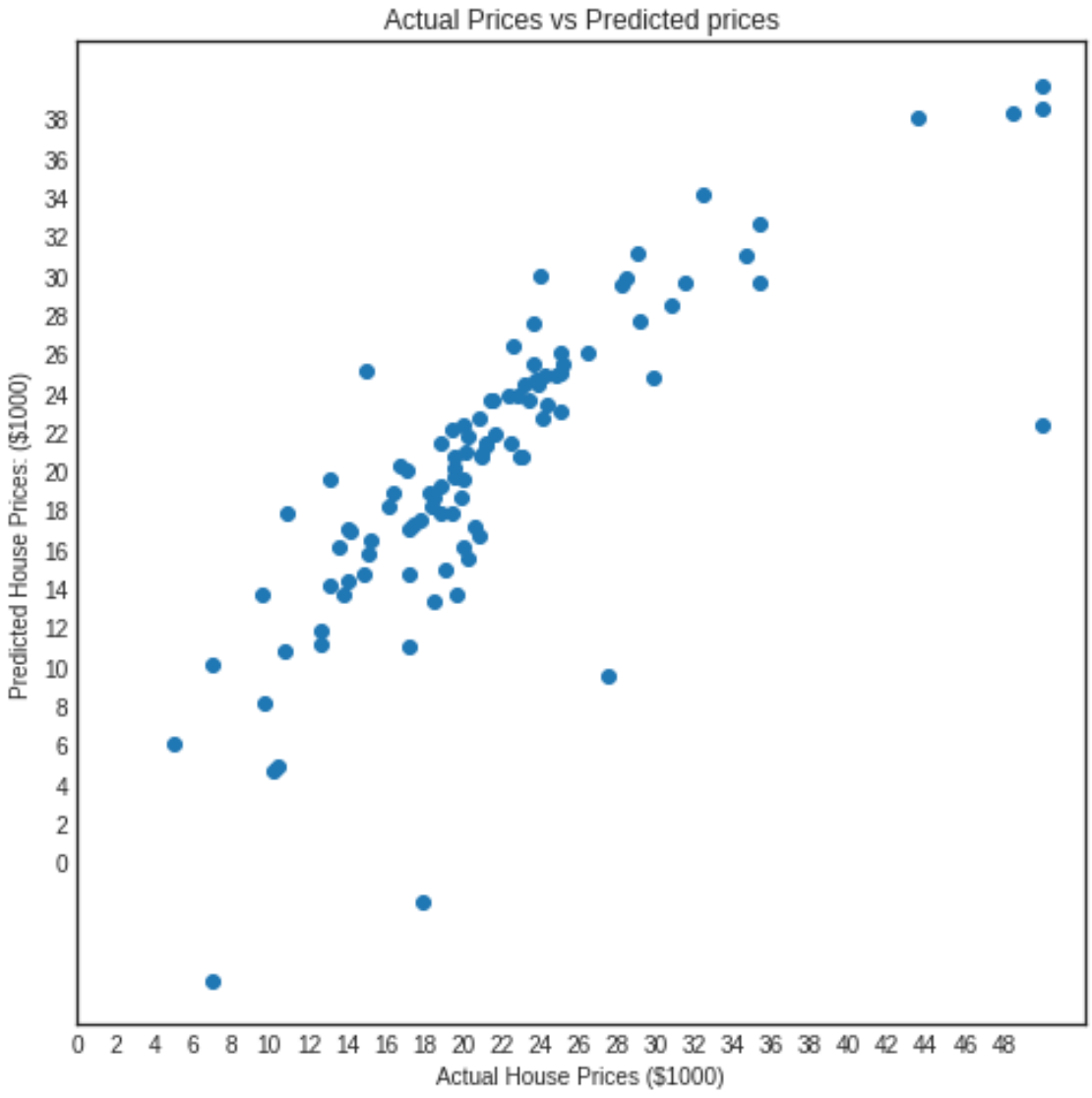
```
Initial cost is:  [[303.27769802]]

Optimal parameters are:
 [[-0.71616295]
 [ 0.16638766]
 [-0.30240052]
 [ 0.8580649 ]
 [-0.84697923]
 [ 3.54084994]
 [-0.13872978]
 [-1.79277331]
 [ 0.53246683]
 [-0.38909839]
 [-1.76000144]
 [ 1.08676187]
 [-3.31181398]
 [22.14352812]]

Final cost is:  11.509475157548042
```



[24]:
```
# Predicted prices
y_test_pred = predict(x_test, w_opt)
```

[25]:
```
plt.figure(figsize=(8, 8))
plt.scatter(y_test, y_test_pred)
plt.xlabel("Actual House Prices ($1000)")
plt.ylabel("Predicted House Prices: ($1000)")
plt.xticks(range(0, int(max(y_test)), 2))
plt.yticks(range(0, int(max(y_test_pred)), 2))
plt.title("Actual Prices vs Predicted prices")
```

[25]:
```
Text(0.5, 1.0, 'Actual Prices vs Predicted prices')
```



9 Performance evaluation

To evaluate the performance of our linear model, we define the following score:

$$Score = 1 - \frac{\sum_{i=1}^{q}(y^{(i)} - \hat{y}^{(i)})^2}{\sum_{i=1}^{q}(y^{(i)} - \overline{y}^{(i)})^2} \tag{8}$$ (9)

We also compare our model to the *Scikit-learn* linear regressor.

[27]:
```
def score(y, y_pred):
    score = 1 - (((y - y_pred) ** 2).sum() / ((y - y.mean()) ** 2).sum())
    return score
```

[28]:
```
# Sklearn linear regressor
from sklearn.linear_model import LinearRegression

sklearn_regressor = LinearRegression().fit(x_train, y_train)
sklearn_train_accuracy = sklearn_regressor.score(x_train, y_train)
sklearn_test_accuracy = sklearn_regressor.score(x_test, y_test)

# Prediction for training set
y_train_pred = predict(x_train, w_opt)
train_accuracy = score(y_train, y_train_pred)
test_accuracy = score(y_test[:, np.newaxis], y_test_pred)
print("Training accuracy   Our model -> %f\tSklearn's implementation -> %f" %␣
 ↪(train_accuracy, sklearn_train_accuracy))
print("Test accuracy       Our model -> %f\tSklearn's implementation -> %f" %␣
 ↪(test_accuracy, sklearn_test_accuracy))
```

```
Training accuracy   Our model -> 0.735029       Sklearn's implementation ->
0.750886
Test accuracy       Our model -> 0.625989       Sklearn's implementation ->
0.668759
```

10 Additional 1

Try out different learning rates to converge quickly. Choose different values of the learning rate on a log-scale.

[29]:
```
# Sklearn linear regressor
from sklearn.linear_model import LinearRegression

sklearn_regressor = LinearRegression().fit(x_train, y_train)
sklearn_train_accuracy = sklearn_regressor.score(x_train, y_train)
sklearn_test_accuracy = sklearn_regressor.score(x_test, y_test)

# Prediction for training set
```

```
y_train_pred = predict(x_train, w_opt)
train_accuracy = score(y_train, y_train_pred)
test_accuracy = score(y_test[:, np.newaxis], y_test_pred)
print("Training accuracy   Our model -> %f\tSklearn's implementation -> %f" %␣
 ↪(train_accuracy, sklearn_train_accuracy))
print("Test accuracy       Our model -> %f\tSklearn's implementation -> %f" %␣
 ↪(test_accuracy, sklearn_test_accuracy))
```

```
Training accuracy   Our model -> 0.735029       Sklearn's implementation ->
0.750886
Test accuracy       Our model -> 0.625989       Sklearn's implementation ->
0.668759
```

11 Additional 2

Linear regression problems can also be solved (in one step) in closed-form using the following formula:

$w = (X^T X)^{-1} X^T y.$

Such formula does not require any feature scaling.

[30]:
```
#####################################################
# Loss function (mean squared error -> MSE)
#####################################################

def one_step_formula(x, y):
    """
    Inputs:
        x: np.ndarray input data of shape [num_samples, num_feat + 1]
        y: np.ndarray targets data of shape [num_samples, 1]
    Outputs:
        weights
    """
    inverse = np.linalg.inv((np.dot(x.T, x)))
    return np.dot(np.dot(inverse, x.T), y)
```

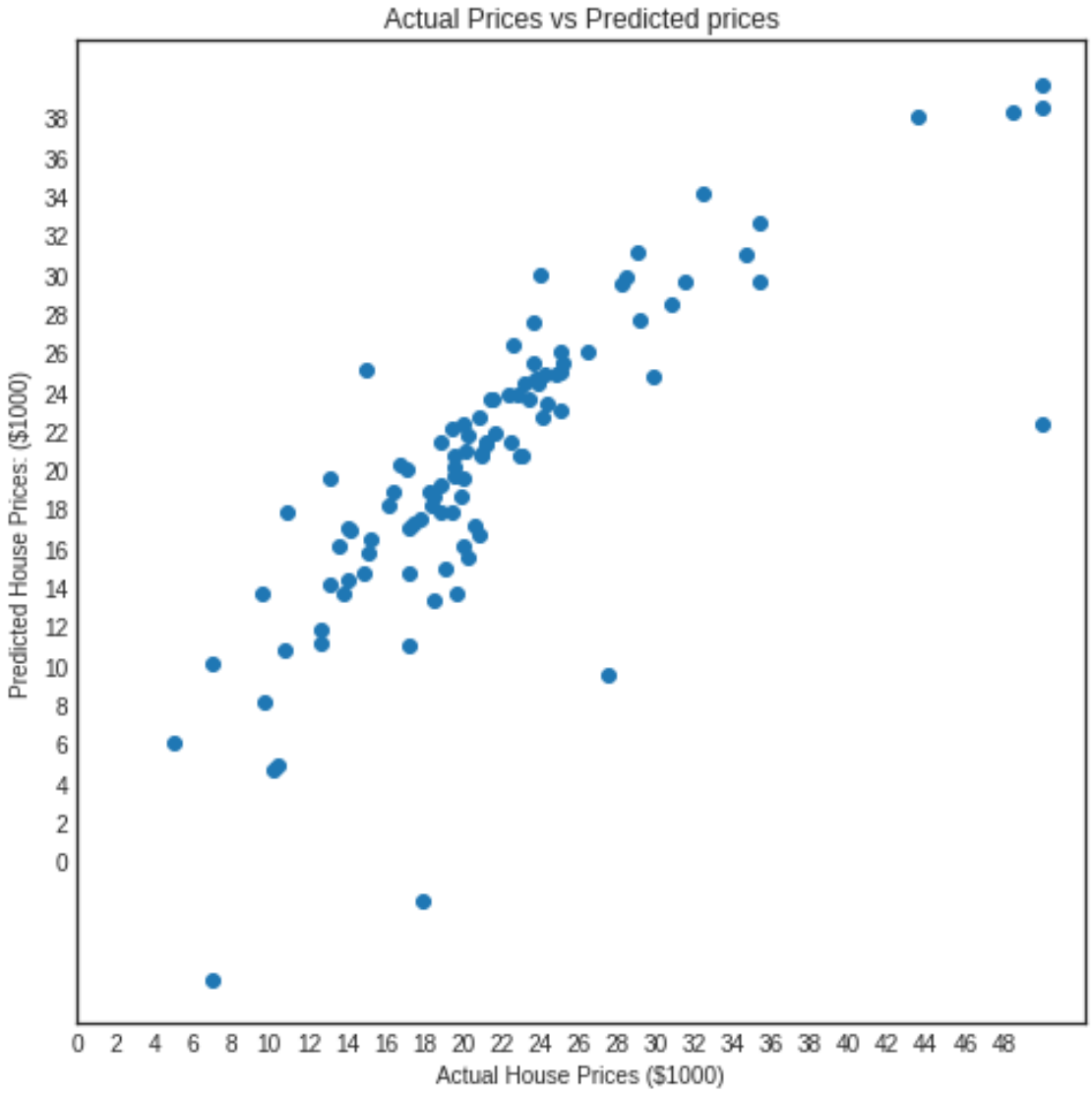
[31]:
```
# Parameters for the gradient descent
num_iters = 3000

# Compute the initial cost
initial_cost = compute_cost(x_train, y_train, w)
print("Initial cost is: ", initial_cost)

# Apply gradient descent algorithm
w = one_step_formula(x_train, y_train)
y_predicted = predict(x_train, w)
final_cost = compute_cost(x_train, y_predicted, w)
```

```
print("Final cost", final_cost)
print("Optimal parameters are: \n", w)
print("Final cost is: ", final_cost)
```

```
Initial cost is:  303.27769801980196
Final cost 0.0
Optimal parameters are:
 [[-1.00213533]
 [ 0.69626862]
 [ 0.27806485]
 [ 0.7187384 ]
 [-2.0223194 ]
 [ 3.14523956]
 [-0.17604788]
 [-3.0819076 ]
 [ 2.25140666]
 [-1.76701378]
 [-2.03775151]
 [ 1.12956831]
 [-3.61165842]
 [22.79653465]]
Final cost is:  0.0
```

[32]:
```
y_test_predicted = predict(x_train, w)
plt.figure(figsize=(8, 8))
plt.scatter(y_test, y_test_pred)
plt.xlabel("Actual House Prices ($1000)")
plt.ylabel("Predicted House Prices: ($1000)")
plt.xticks(range(0, int(max(y_test)), 2))
plt.yticks(range(0, int(max(y_test_pred)), 2))
plt.title("Actual Prices vs Predicted prices")
```

[32]:
```
Text(0.5, 1.0, 'Actual Prices vs Predicted prices')
```



[33]:
```
# Sklearn linear regressor
from sklearn.linear_model import LinearRegression

sklearn_regressor = LinearRegression().fit(x_train, y_train)
sklearn_train_accuracy = sklearn_regressor.score(x_train, y_train)
sklearn_test_accuracy = sklearn_regressor.score(x_test, y_test)

# Prediction for training set
y_train_pred = predict(x_train, w)
train_accuracy = score(y_train, y_train_pred)
test_accuracy = score(y_test[:, np.newaxis], y_test_pred)
print("Training accuracy   Our model -> %f\tSklearn's implementation -> %f" %␣
 ↪(train_accuracy, sklearn_train_accuracy))
```

```
print("Test accuracy       Our model -> %f\tSklearn's implementation -> %f" %␣
 ↪(test_accuracy, sklearn_test_accuracy))
```

```
Training accuracy   Our model -> 0.750886       Sklearn's implementation ->
0.750886
Test accuracy       Our model -> 0.625989       Sklearn's implementation ->
0.668759
```

```
[ ]:
```