CAIS++ Linear Regression Workshop

Before you go through this code, make sure you read Lesson 2 (http://caisplusplus.usc.edu/blog/curriculum/lesson2) from our curriculum!

Part 1: Importing the Data

In [31]:
```
##importing numpy and the boston data set:

import numpy as np
from sklearn.datasets import load_boston
```

In [32]:
```
boston = load_boston()
print(boston.keys())
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR'])
```

In [33]:
```python
print(boston.DESCR)
```

```
Boston House Prices dataset
===========================

Notes
------
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 rive
r; 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.
http://archive.ics.uci.edu/ml/datasets/Housing (http://archive.ics.uci.ed
u/ml/datasets/Housing)


This dataset was taken from the StatLib library which is maintained at Ca
rnegie 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 diagnos
tics
...', Wiley, 1980.   N.B. Various transformations are used in the table o
n
pages 244-261 of the latter.

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

```
**References**

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influent
ial Data and Sources of Collinearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learnin
g. In Proceedings on the Tenth International Conference of Machine Learni
ng, 236-243, University of Massachusetts, Amherst. Morgan Kaufmann.
   - many more! (see http://archive.ics.uci.edu/ml/datasets/Housing) (htt
p://archive.ics.uci.edu/ml/datasets/Housing))
```

In [34]:

```python
# Investigate shape of the input data array
data = boston.data
target = boston.target ## according to the description above, the target is

print(data.shape)
print(target.shape)
print(boston.feature_names)

num_features = len(boston.feature_names) #13 features
num_samples = data.shape[0] # 506 training examples
```

```
(506, 13)
(506,)
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

In [35]:

```python
# Use Pandas to get an overview of the training data

import pandas as pd
bos_dataframe = pd.DataFrame(boston.data)
bos_dataframe.columns = boston.feature_names
bos_dataframe.head()
```

Out[35]:

	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 [36]:
```
# Add in the target variable: price

bos_dataframe['PRICE'] = target
bos_dataframe.head()
```

Out[36]:

	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

Part 2: Setting up the Machine Learning Objective

In [37]:
```
# 1. Randomly initalize a weights vector between (-1,  1). Keep in mind: wha
# 2. Call it weights_init.
# 3. Print weights_init
#############################################
weights_init = np.random.uniform(-1,1, num_features)
print(weights_init)
```

```
[ 0.05816157  0.26497918  0.30575397  0.98813133 -0.98049133  0.94014403
 -0.45712372  0.73179466  0.65141797 -0.27228182 -0.50943419 -0.41859358
  0.92235852]
```

In [38]:
```
# Create a variable for the bias, called bias_init.Initalize the bias to 0
#############################################
bias_init = 0
```

2.1: Normalize the input data. We do this because so that we can get all of our data in the same scale.

More information can be found [here (https://stats.stackexchange.com/questions/41704/how-and-why-do-normalization-and-feature-scaling-work)](https://stats.stackexchange.com/questions/41704/how-and-why-do-normalization-and-feature-scaling-work)

In [40]:
```
# 1. For each feature (coloumn in the data set), calculate the mean and the
# 2. For each data point in that feature, subtract the mean and then divide

# (uncomment below, and complete for loop)
# for i in range(num_features):

#average_CRIM = np.mean.CRIM
for i in range(num_features):
    feature_avg = np.mean(data[:,i])
    feature_max = np.amax(data[:,i])
    data[:,i] = ((data[:,i] - feature_avg) /feature_max)

# now the values should be normalized (uncomment below):
# bos_dataframe.head()
```

2.2 Defining the hypothesis and the cost function:

The Hypothesis function returns a vector of predicted prices.

1. Since we are working with multiple features, we need to dot product the input data with the weights vector. Use the numpy dot() function!
$$h_w(x) = w_0 + w_1x_1 + w_2x_2 + \ldots + w_nx_n$$
2. Now we need to add our bias to each input value. use numpy's repeat function to create a vector of length 'num_samples' of the bias_init.
3. Return the dot product of the input data and weights summed with the bias vector.

The function header has been defined for you, but you need to complete it.

In [43]:
```
def hypothesis(weights, bias):

    return data.dot(weights) + bias
```

In [44]:
```
# Run this cell to see the shape of the return value of the hypothesis funct
# (BONUS: try to think of what the shape would be before printing it out)
hypothesis(weights_init, bias_init).shape
```

Out[44]:
```
(506,)
```

1. Define the cost function, which is just subtracting the actual target from our hypothesis, and squaring (use np.square()) that error.
2. We then take the mean (use np.mean()) of all these squared errors. Remember that we dvide by 2 to make the math easier later on:

$$MSE\ Cost = J(w_0, w_1) = \frac{1}{2m}\sum_{i=0}^{m}(h_w(x^{(i)}) - y^{(i)})^2$$

3. The function header has been defined for you again, but you need to complete it:

In [45]:
```
def cost(weights, bias):
    data_error = np.square(hypothesis(weights, bias) - target)
    return np.mean(data_error)/2
```

In [46]:
```
# Run this cell to print out the inital cost. It's really large right now!
hypothesis(weights_init, bias_init).shape
cost(weights_init, bias_init)
```

Out[46]:
```
295.9449330783788
```

The gradient function has been defined for you. It calculates the partial derivative for the weights and bias (look at the red and blue rectangles:

repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \boxed{\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right)}$$

$$\theta_1 := \theta_1 - \alpha \boxed{\frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(x^{(i)}) - y^{(i)}\right) \cdot x^{(i)}}$$

}

In [47]:
```
# Gradient: return weight gradient vector, bias gradient at current step

def gradient(weights, bias):
    weight_gradients = []

    for (weight_num, weight) in enumerate(weights):
        grad = np.mean((hypothesis(weights, bias)-target) * data[:, weight_n
        weight_gradients.append(grad)

    weight_gradients = np.array(weight_gradients)

    bias_gradient = np.mean(hypothesis(weights, bias) - target)

    return (weight_gradients, bias_gradient)
```

In [48]:
```
# Check to make sure it works
# Initial gradient should be large

gradient(weights_init, bias_init)
```

Out[48]:
```
(array([ 0.34755494, -0.74000449,  1.09133458, -0.35368968,  0.51346534,
        -0.50667737,  0.94385727, -0.37849993,  1.30423226,  1.03163903,
         0.4560386 , -0.72745941,  1.27634343]), -22.532806324110673)
```

2.3: Run Gradient Descent

1. You want to update the weights by subtracting the partial derivative * some learning rate alpha.
2. Do the same for the bias
3. Append the cost of the new weights and bias to an array of costs using np.append()
4. Repeat for some number (we call this the number of epochs, or iterations of steps we're completing during gradient descent)
5. As always, the function header is defined for you. Complete the rest!

In [*]:
```
# Gradient descent algorithm:
# Repeat for desired iterations: Calculate gradient, move down one step
# Cost should decrease over time

LEARNING_RATE = 0.01

def gradient_descent(weights, bias, num_epochs):
    costs = []
    weights = weights
    bias = bias

    for i in range(num_epochs):
        weights_gradient, bias_gradient = gradient(weights, bias)

        # write your code here:
        weights = weights - LEARNING_RATE * weights_gradient
        bias = bias - LEARNING_RATE * bias_gradient
        costs.append(cost(weights,bias))


    return costs, weights, bias
```

In [50]:
```
costs, trained_weights, trained_bias = gradient_descent(weights_init, bias_i
```

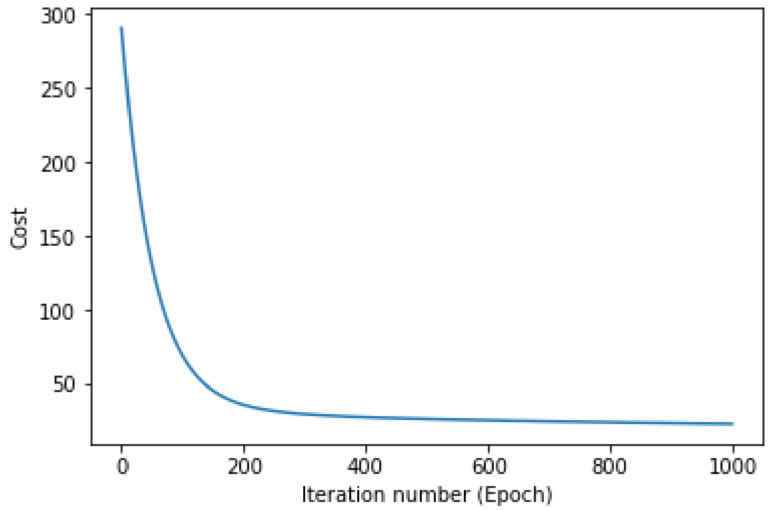
In [51]:
```
print(trained_weights)
print(trained_bias)
```

```
[-1.36943946  2.84281467 -3.31898027  3.56546457 -2.38743623  4.65480604
 -2.60309806  0.15588501 -1.90298105 -3.38011495 -3.24532319  2.43247369
 -6.53640968]
22.53183355475399
```

Part 4: Evaluating the Model

In [52]:
```
import matplotlib.pyplot as plt
```

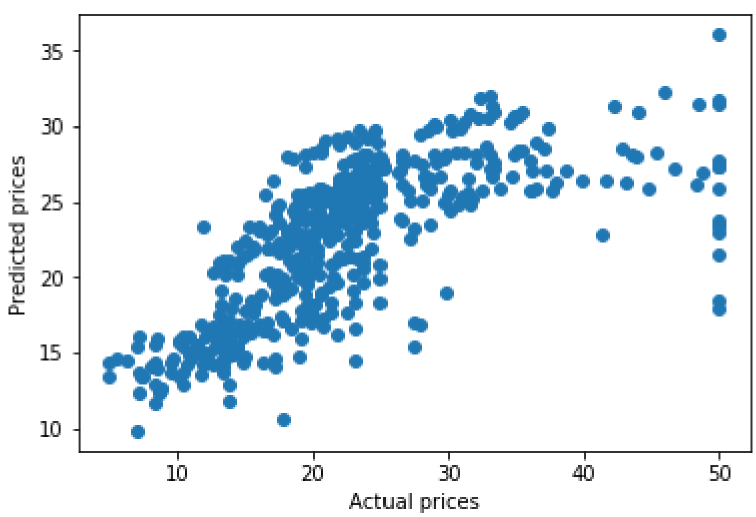
In [53]:
```
plt.plot(costs)
plt.xlabel("Iteration number (Epoch)")
plt.ylabel("Cost")
plt.show()
```



In [54]:
```
# Final predicted prices
new_hypotheses = hypothesis(trained_weights, trained_bias)
```

In [56]:
```
# Make sure predictions, actual values are correlated

plt.scatter(target, new_hypotheses)
plt.xlabel("Actual prices")
plt.ylabel("Predicted prices")
plt.show()
```



Congrats! You just did machine learning

Part 5: Using sklearn's built-in linear regression

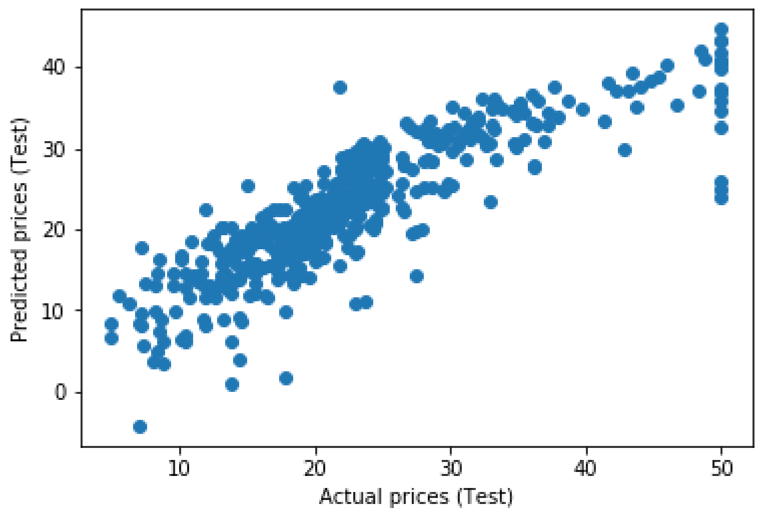
functionality:

In [57]:
```
from sklearn import linear_model
regr = linear_model.LinearRegression()
```

In [60]:
```
## call the .fit() function on regr using data and target. Yes it's that eas
###################################
regr.fit(data,target)
```

Out[60]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=Fal
se)
```

In [61]:
```
plt.scatter(target, regr.predict(data))
plt.xlabel("Actual prices (Test)")
plt.ylabel("Predicted prices (Test)")
plt.show()
```



Train Test Split:

What we often do in machine learning is split our data into a training set and a testing set. This is so that once we train our model on our training set, we aren't making predictions on the same input, as that would give us "too-good" answers, so instead we put aside some data into a testing set and make predictions on that once we've trained our model

In [ ]:
```
from sklearn.model_selection import train_test_split

## Using sklearn's train_test_split() function, create 4 variables X_train,
## For function parameters, the test size will be 0.25, and the random_state
## Print each of these variables:
```

In [ ]:
```
## use .fit() to train the regression model below
```

In [ ]:
```
plt.scatter(Y_test, regr.predict(X_test))
plt.xlabel("Actual prices (Test)")
plt.ylabel("Predicted prices (Test)")
plt.show()
```

In [ ]:
```
```