Regression in Python

This is a very quick run-through of some basic statistical concepts, adapted from Lab 4 in Harvard's CS109 (https://github.com/cs109/2015lab4) course. Please feel free to try the original lab if you're feeling ambitious :-) The CS109 git repository also has the solutions if you're stuck.

- Linear Regression Models
- Prediction using linear regression
- Some re-sampling methods
  - Train-Test splits
  - Cross Validation

Linear regression is used to model and predict continuous outcomes while logistic regression is used to model binary outcomes. We'll see some examples of linear regression as well as Train-test splits.

The packages we'll cover are: `statsmodels`, `seaborn`, and `scikit-learn`. While we don't explicitly teach `statsmodels` and `seaborn` in the Springboard workshop, those are great libraries to know.

(https://imgs.xkcd.com/comics/sustainable.png)

In [2]:
```
# special IPython command to prepare the notebook for matplotlib and other lib
raries
%pylab inline

import numpy as np
import pandas as pd
import scipy.stats as stats
import matplotlib.pyplot as plt
import sklearn

import seaborn as sns

# special matplotlib argument for improved plots
from matplotlib import rcParams
sns.set_style("whitegrid")
sns.set_context("poster")
```

```
Populating the interactive namespace from numpy and matplotlib
```

Part 1: Linear Regression

Purpose of linear regression

Given a dataset X and Y, linear regression can be used to:

- Build a **predictive model** to predict future values of X_i without a Y value.
- Model the **strength of the relationship** between each dependent variable X_i and Y
  - Sometimes not all X_i will have a relationship with Y
  - Need to figure out which X_i contributes most information to determine Y
- Linear regression is used in so many applications that I won't warrant this with examples. It is in many cases, the first pass prediction algorithm for continuous outcomes.

A brief recap (feel free to skip if you don't care about the math)

Linear Regression (http://en.wikipedia.org/wiki/Linear_regression) is a method to model the relationship between a set of independent variables X (also knowns as explanatory variables, features, predictors) and a dependent variable Y. This method assumes the relationship between each predictor X is linearly related to the dependent variable Y.

$$Y = \beta_0 + \beta_1 X + \epsilon$$

where ϵ is considered as an unobservable random variable that adds noise to the linear relationship. This is the simplest form of linear regression (one variable), we'll call this the simple model.

- β_0 is the intercept of the linear model
- Multiple linear regression is when you have more than one independent variable
  - $X_1, X_2, X_3, \ldots$

$$Y = \beta_0 + \beta_1 X_1 + \ldots + \beta_p X_p + \epsilon$$

- Back to the simple model. The model in linear regression is the *conditional mean* of Y given the values in X is expressed a linear function.

$$y = f(x) = E(Y|X = x)$$

http://www.learner.org/courses/againstallodds/about/glossary.html (http://www.learner.org/courses/againstallodds/about/glossary.html)

- The goal is to estimate the coefficients (e.g. β_0 and β_1). We represent the estimates of the coefficients with a "hat" on top of the letter.

$$\hat{\beta}_0, \hat{\beta}_1$$

- Once you estimate the coefficients $\hat{\beta}_0$ and $\hat{\beta}_1$, you can use these to predict new values of Y

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1$$

- How do you estimate the coefficients?
  - There are many ways to fit a linear regression model
  - The method called **least squares** is one of the most common methods
  - We will discuss least squares today

Estimating $\hat{\beta}$: Least squares

Least squares (http://en.wikipedia.org/wiki/Least_squares) is a method that can estimate the coefficients of a linear model by minimizing the difference between the following:

$$S = \sum_{i=1}^{N} r_i = \sum_{i=1}^{N} (y_i - (\beta_0 + \beta_1 x_i))^2$$

where N is the number of observations.

- We will not go into the mathematical details, but the least squares estimates $\hat{\beta}_0$ and $\hat{\beta}_1$ minimize the sum of the squared residuals $r_i = y_i - (\beta_0 + \beta_1 x_i)$ in the model (i.e. makes the difference between the observed y_i and linear model $\beta_0 + \beta_1 x_i$ as small as possible).

The solution can be written in compact matrix notation as

$$\hat{\beta} = (X^T X)^{-1} X^T Y$$

We wanted to show you this in case you remember linear algebra, in order for this solution to exist we need $X^T X$ to be invertible. Of course this requires a few extra assumptions, X must be full rank so that $X^T X$ is invertible, etc. **This is important for us because this means that having redundant features in our regression models will lead to poorly fitting (and unstable) models.** We'll see an implementation of this in the extra linear regression example.

Note: The "hat" means it is an estimate of the coefficient.

Part 2: Boston Housing Data Set

The Boston Housing data set (https://archive.ics.uci.edu/ml/datasets/Housing) contains information about the housing values in suburbs of Boston. This dataset was originally taken from the StatLib library which is maintained at Carnegie Mellon University and is now available on the UCI Machine Learning Repository.

Load the Boston Housing data set from `sklearn`

This data set is available in the sklearn (http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_boston.html#sklearn.datasets.load_boston) python module which is how we will access it today.

In [3]:
```
from sklearn.datasets import load_boston
boston = load_boston()
```

In [4]:
```
boston.keys()
```

Out[4]:
```
dict_keys(['data', 'target', 'feature_names', 'DESCR'])
```

In [5]:
```
boston.data.shape
```

Out[5]:
```
(506, 13)
```

In [6]:
```
# Print column names
print(boston.feature_names)
```

```
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

In [7]:
```
# Print description of Boston housing data set
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 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.
http://archive.ics.uci.edu/ml/datasets/Housing


This dataset was taken from the StatLib library which is maintained at Carneg
ie 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 tha
t address regression
problems.

**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, 236-24
3, University of Massachusetts, Amherst. Morgan Kaufmann.
   - many more! (see http://archive.ics.uci.edu/ml/datasets/Housing)
```

Now let's explore the data set itself.

In [8]:
```
bos = pd.DataFrame(boston.data)
bos.head()
```

Out[8]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
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

There are no column names in the DataFrame. Let's add those.

In [9]:
```
bos.columns = boston.feature_names
bos.head()
```

Out[9]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
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
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
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
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
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

Now we have a pandas DataFrame called `bos` containing all the data we want to use to predict Boston Housing prices. Let's create a variable called `PRICE` which will contain the prices. This information is contained in the `target` data.

In [10]:
```
print(boston.target.shape)
```

```
(506,)
```

In [11]:
```
bos['PRICE'] = boston.target
bos.head()
```

Out[11]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B
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
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
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
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
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

EDA and Summary Statistics

Let's explore this data set. First we use `describe()` to get basic summary statistics for each of the columns.

In [12]:
```
bos.describe()
```

Out[12]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.00
mean	3.593761	11.363636	11.136779	0.069170	0.554695	6.284634	68.574
std	8.596783	23.322453	6.860353	0.253994	0.115878	0.702617	28.148
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900(
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500
75%	3.647423	12.500000	18.100000	0.000000	0.624000	6.623500	94.075
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.00

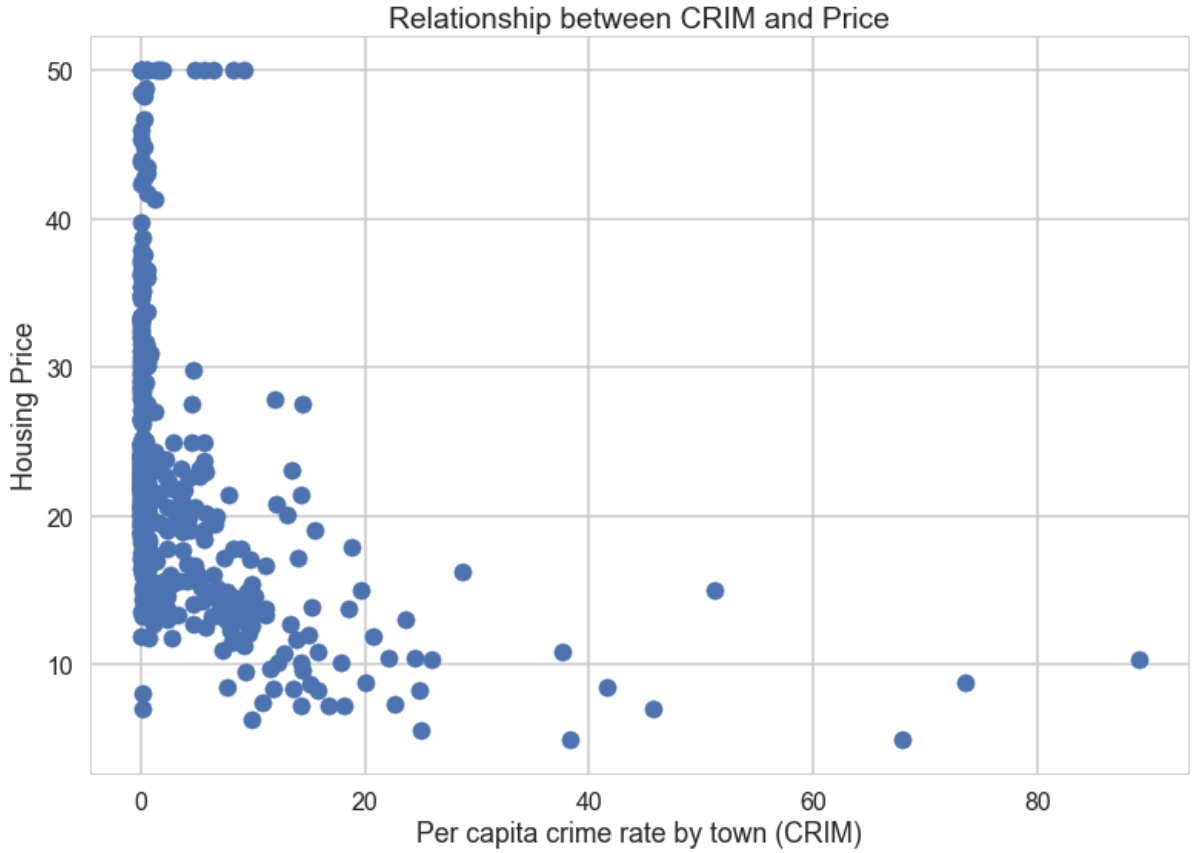
Scatter plots

Let's look at some scatter plots for three variables: 'CRIM', 'RM' and 'PTRATIO'.

What kind of relationship do you see? e.g. positive, negative? linear? non-linear?

In [13]:
```
plt.scatter(bos.CRIM, bos.PRICE)
plt.xlabel("Per capita crime rate by town (CRIM)")
plt.ylabel("Housing Price")
plt.title("Relationship between CRIM and Price")
```

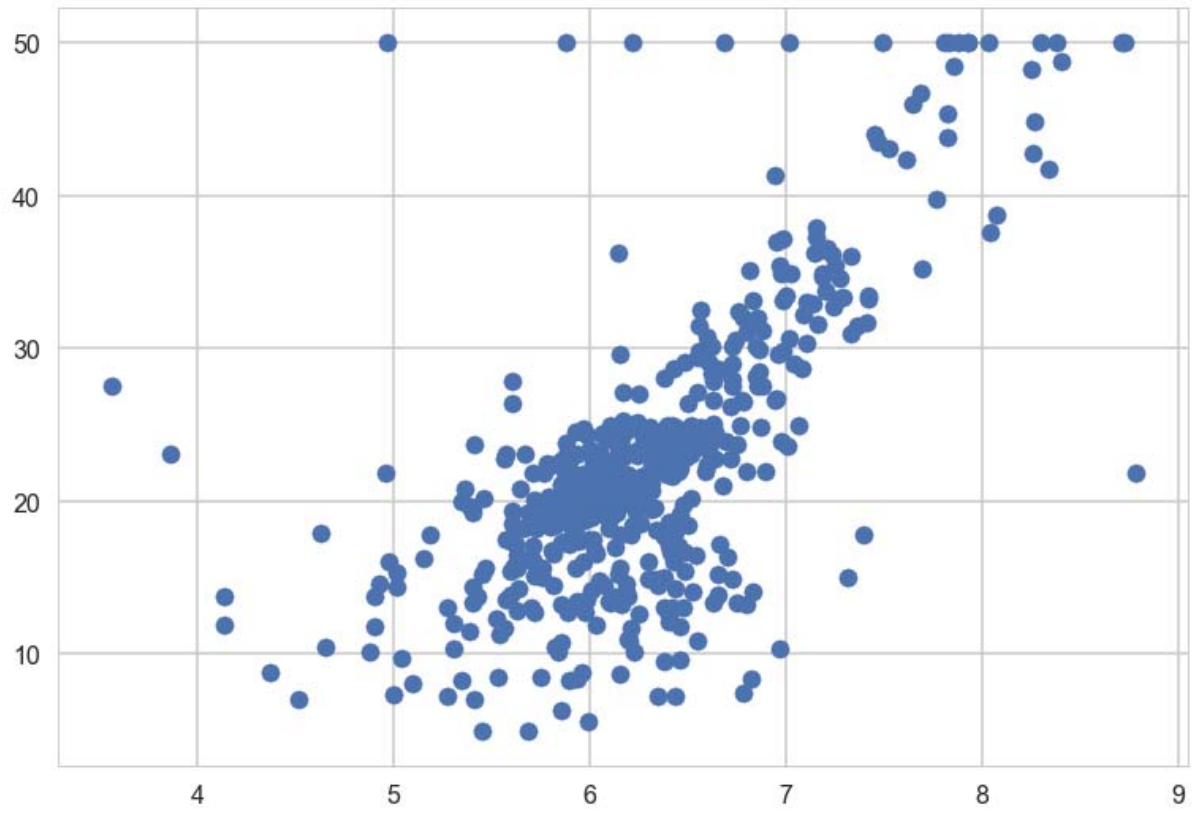
Out[13]: <matplotlib.text.Text at 0xa8dc160>



Your turn: Create scatter plots between *RM* and *PRICE*, and *PTRATIO* and *PRICE*. What do you notice?

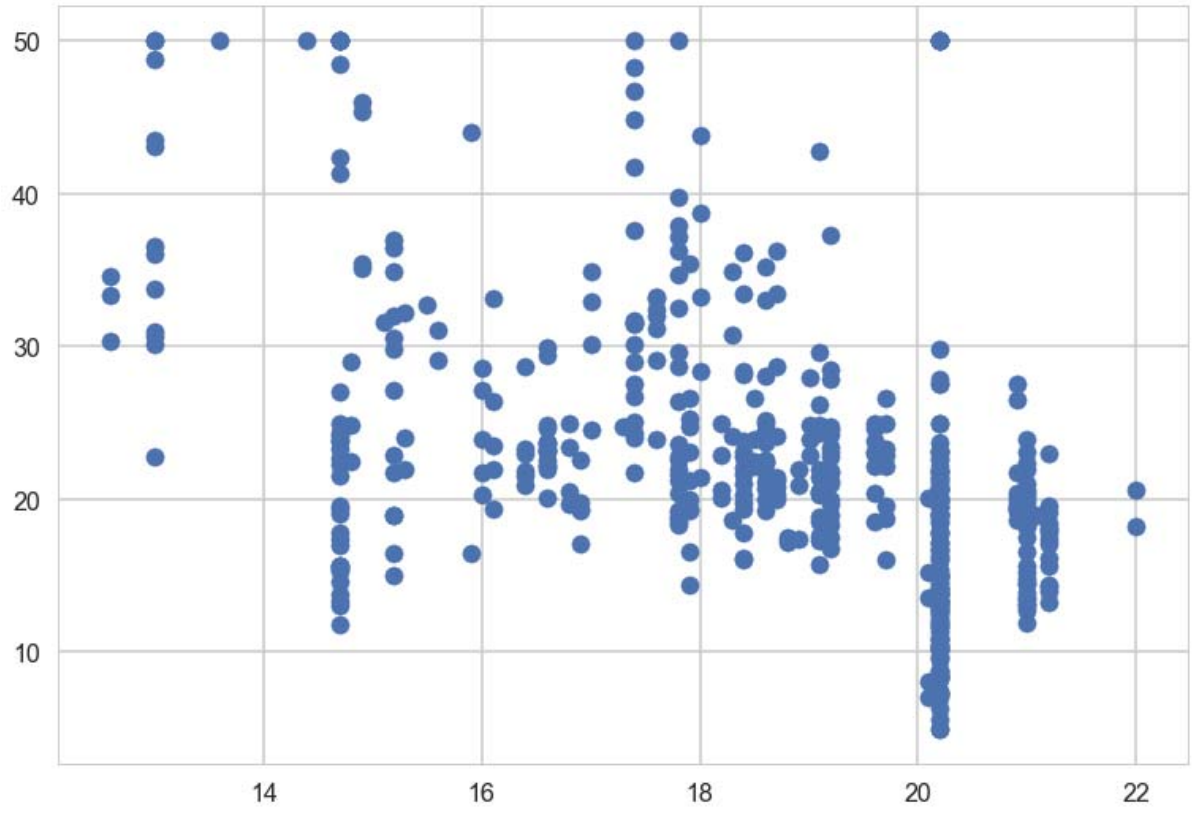
In [14]:
```
#your turn: scatter plot between *RM* and *PRICE*
plt.scatter(bos.RM, bos.PRICE)
```

Out[14]: <matplotlib.collections.PathCollection at 0xabfa908>



In [15]:
```
#your turn: scatter plot between *PTRATIO* and *PRICE*
plt.scatter(bos.PTRATIO, bos.PRICE)
```

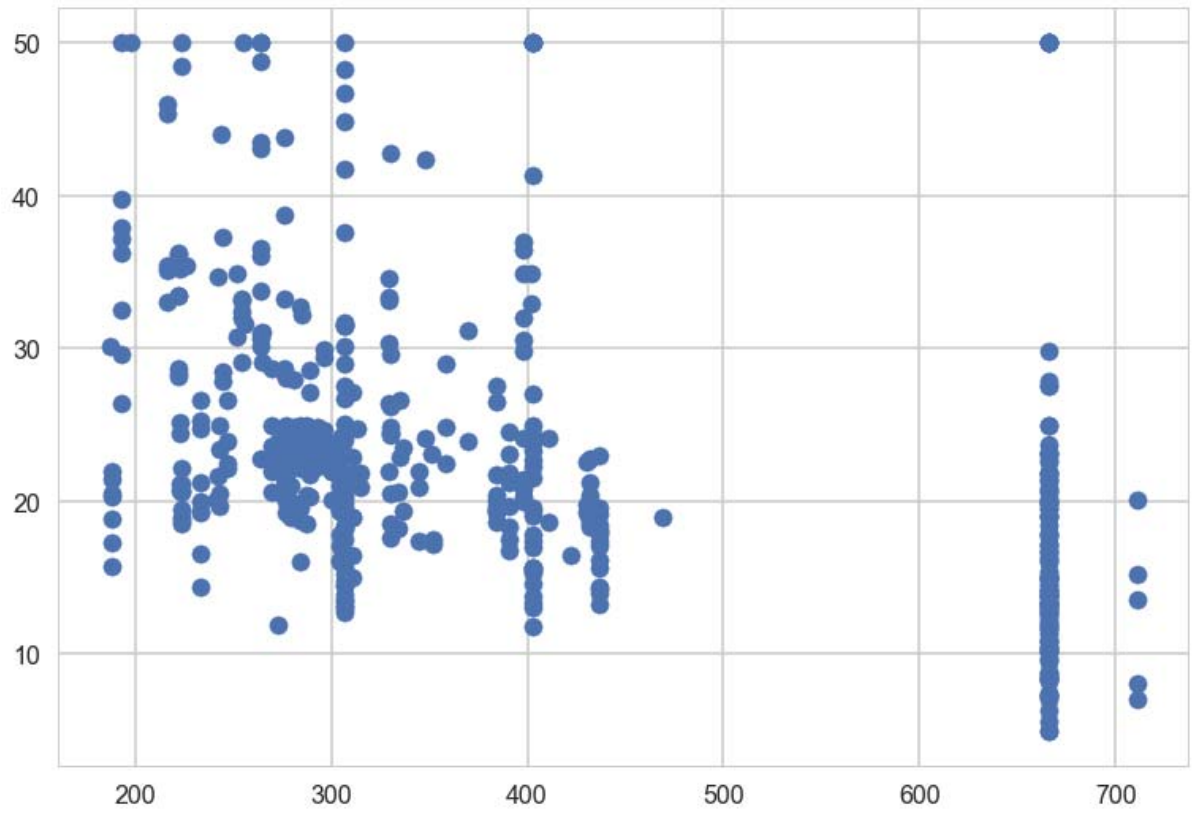
Out[15]: <matplotlib.collections.PathCollection at 0xafc5940>



Your turn: What are some other numeric variables of interest? Plot scatter plots with these variables and *PRICE*.

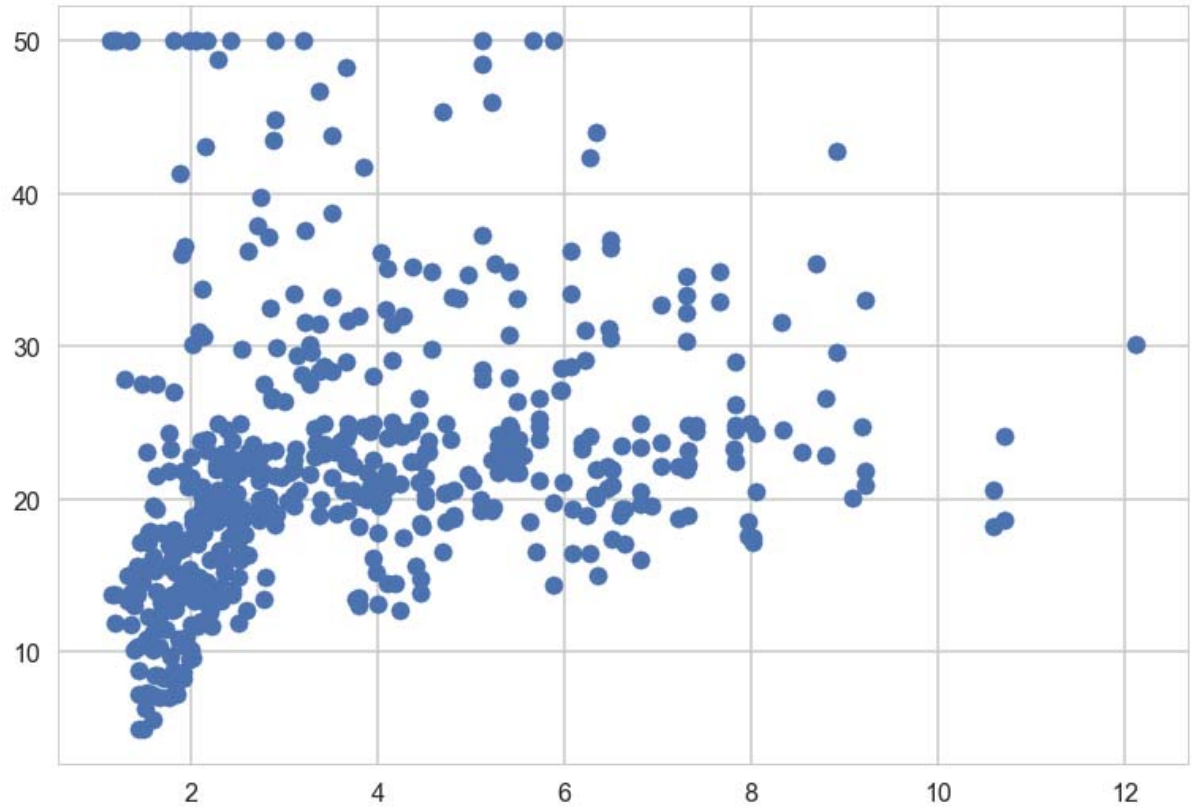
In [16]:
```
#your turn: create some other scatter plots
plt.scatter(bos.TAX, bos.PRICE)
```

Out[16]: <matplotlib.collections.PathCollection at 0xb0137b8>



In [17]:
```
plt.scatter(bos.DIS, bos.PRICE)
```

Out[17]:
```
<matplotlib.collections.PathCollection at 0xc3b2b70>
```



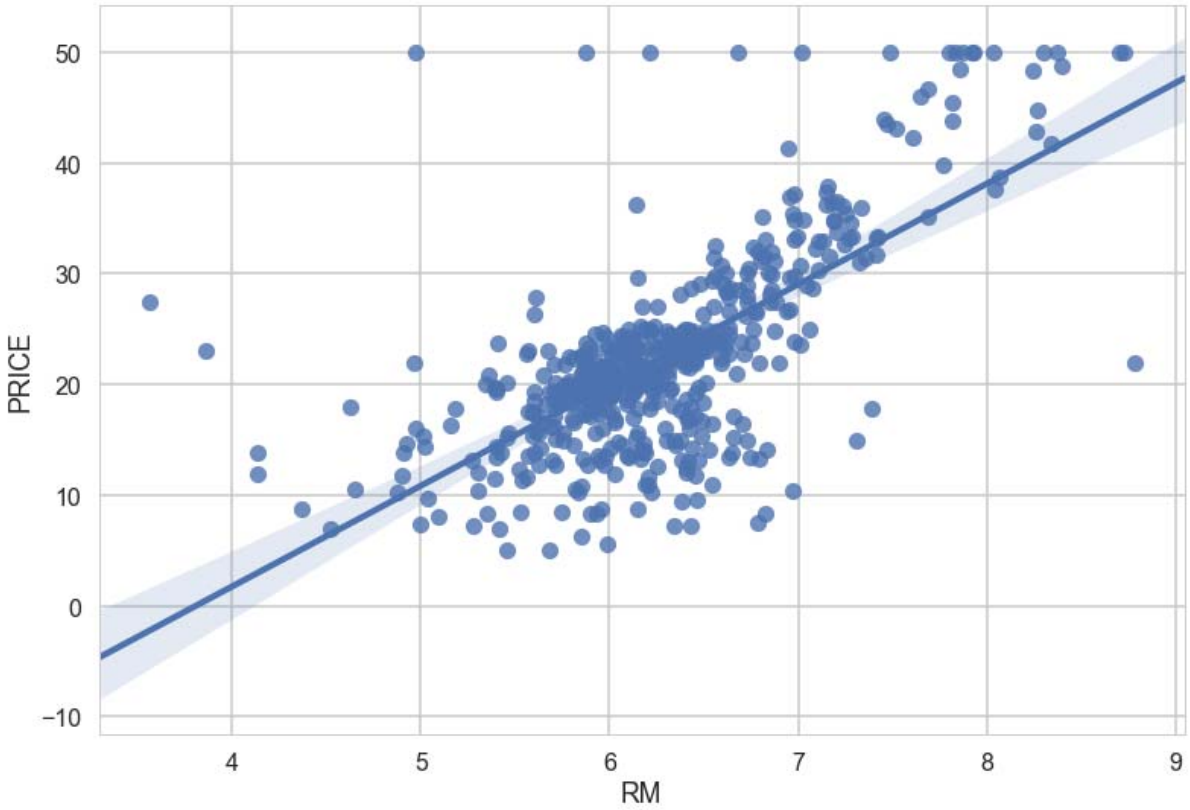
Scatter Plots using Seaborn

Seaborn (https://stanford.edu/~mwaskom/software/seaborn/) is a cool Python plotting library built on top of matplotlib. It provides convenient syntax and shortcuts for many common types of plots, along with better-looking defaults.

We can also use seaborn regplot (https://stanford.edu/~mwaskom/software/seaborn/tutorial/regression.html#functions-to-draw-linear-regression-models) for the scatterplot above. This provides automatic linear regression fits (useful for data exploration later on). Here's one example below.

In [18]:
```
sns.regplot(y="PRICE", x="RM", data=bos, fit_reg = True)
```

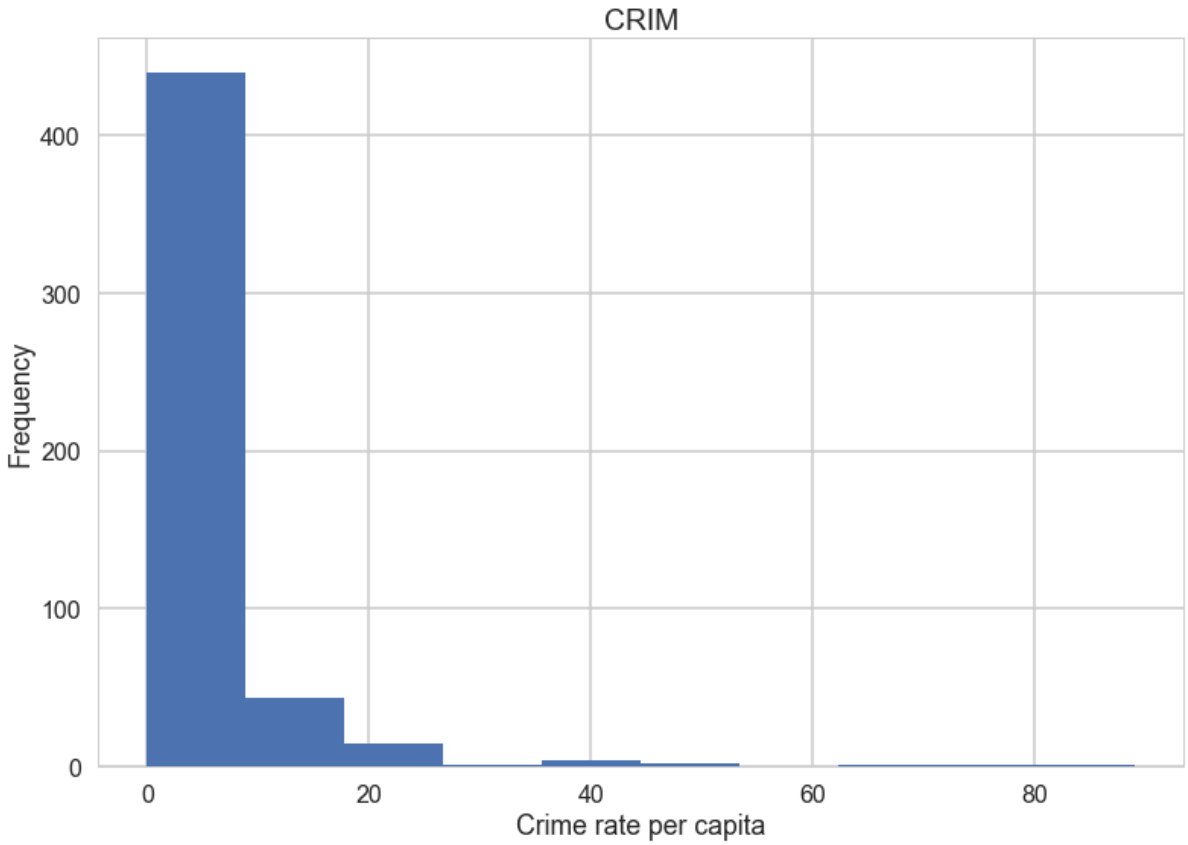
Out[18]:
```
<matplotlib.axes._subplots.AxesSubplot at 0xb1ebfd0>
```



Histograms

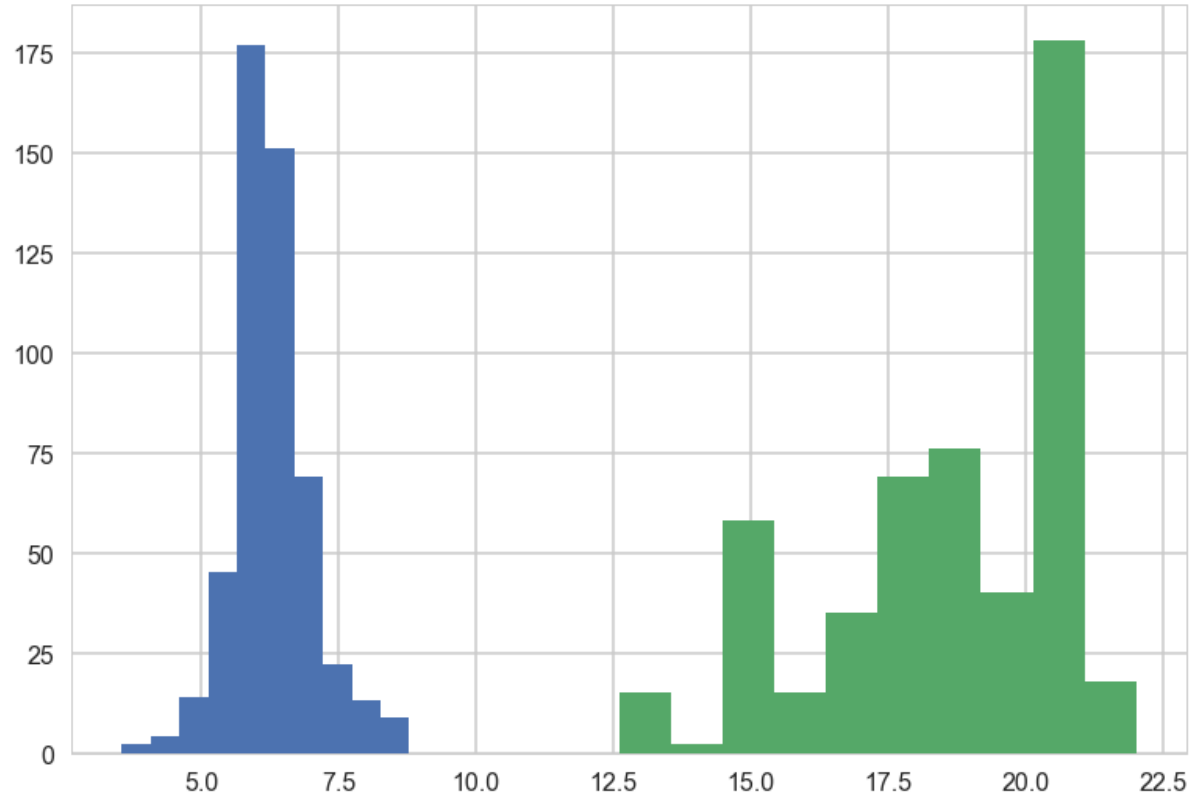
Histograms are a useful way to visually summarize the statistical properties of numeric variables. They can give you an idea of the mean and the spread of the variables as well as outliers.

In [19]:
```
plt.hist(bos.CRIM)
plt.title("CRIM")
plt.xlabel("Crime rate per capita")
plt.ylabel("Frequency")
plt.show()
```



Your turn: Plot separate histograms and one for *RM*, one for *PTRATIO*. Any interesting observations?

In [20]:
```
#your turn
plt.hist(bos.RM)
plt.hist(bos.PTRATIO)
plt.show()
```



Linear regression with Boston housing data example

Here,

Y = boston housing prices (also called "target" data in python)

and

X = all the other features (or independent variables)

which we will use to fit a linear regression model and predict Boston housing prices. We will use the least squares method as the way to estimate the coefficients.

We'll use two ways of fitting a linear regression. We recommend the first but the second is also powerful in its features.

Fitting Linear Regression using `statsmodels`

Statsmodels (http://statsmodels.sourceforge.net/) is a great Python library for a lot of basic and inferential statistics. It also provides basic regression functions using an R-like syntax, so it's commonly used by statisticians. While we don't cover statsmodels officially in the Data Science Intensive, it's a good library to have in your toolbox. Here's a quick example of what you could do with it.

In [21]:
```
# Import regression modules
# ols - stands for Ordinary least squares, we'll use this
import statsmodels.api as sm
from statsmodels.formula.api import ols
```

In [22]:
```
# statsmodels works nicely with pandas dataframes
# The thing inside the "quotes" is called a formula, a bit on that below
m = ols('PRICE ~ RM',bos).fit()
print(m.summary())
```

```
                            OLS Regression Results

==============================================================================
Dep. Variable:                  PRICE   R-squared:                       0.48
4
Model:                            OLS   Adj. R-squared:                  0.48
3
Method:                 Least Squares   F-statistic:                     471.
8
Date:                Wed, 14 Jun 2017   Prob (F-statistic):           2.49e-7
4
Time:                        14:17:33   Log-Likelihood:                -1673.
1
No. Observations:                 506   AIC:                             335
0.
Df Residuals:                     504   BIC:                             335
9.
Df Model:                           1

Covariance Type:            nonrobust

==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. In
t.]
------------------------------------------------------------------------------
-
Intercept    -34.6706      2.650    -13.084      0.000       -39.877   -29.46
5
RM             9.1021      0.419     21.722      0.000         8.279     9.92
5
==============================================================================
=
Omnibus:                      102.585   Durbin-Watson:                   0.68
4
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              612.44
9
Skew:                           0.726   Prob(JB):                     1.02e-13
3
Kurtosis:                       8.190   Cond. No.                         58.
4
==============================================================================
=

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correc
tly specified.
```

Interpreting coefficients

There is a ton of information in this output. But we'll concentrate on the coefficient table (middle table). We can interpret the `RM` coefficient (9.1021) by first noticing that the p-value (under `P>|t|`) is so small, basically zero. We can interpret the coefficient as, if we compare two groups of towns, one where the average number of rooms is say 5 and the other group is the same except that they all have 6 rooms. For these two groups the average difference in house prices is about 9.1 (in thousands) so about $\$9,100$ difference. The confidence interval fives us a range of plausible values for this difference, about ($\$8,279$, $\$9,925$), deffinitely not chump change.

`statsmodels` formulas

This formula notation will seem familiar to `R` users, but will take some getting used to for people coming from other languages or are new to statistics.

The formula gives instruction for a general structure for a regression call. For `statsmodels` (`ols` or `logit`) calls you need to have a Pandas dataframe with column names that you will add to your formula. In the below example you need a pandas data frame that includes the columns named (`Outcome`, `X1`,`X2`, ...), bbut you don't need to build a new dataframe for every regression. Use the same dataframe with all these things in it. The structure is very simple:

```
Outcome ~ X1
```

But of course we want to to be able to handle more complex models, for example multiple regression is doone like this:

```
Outcome ~ X1 + X2 + X3
```

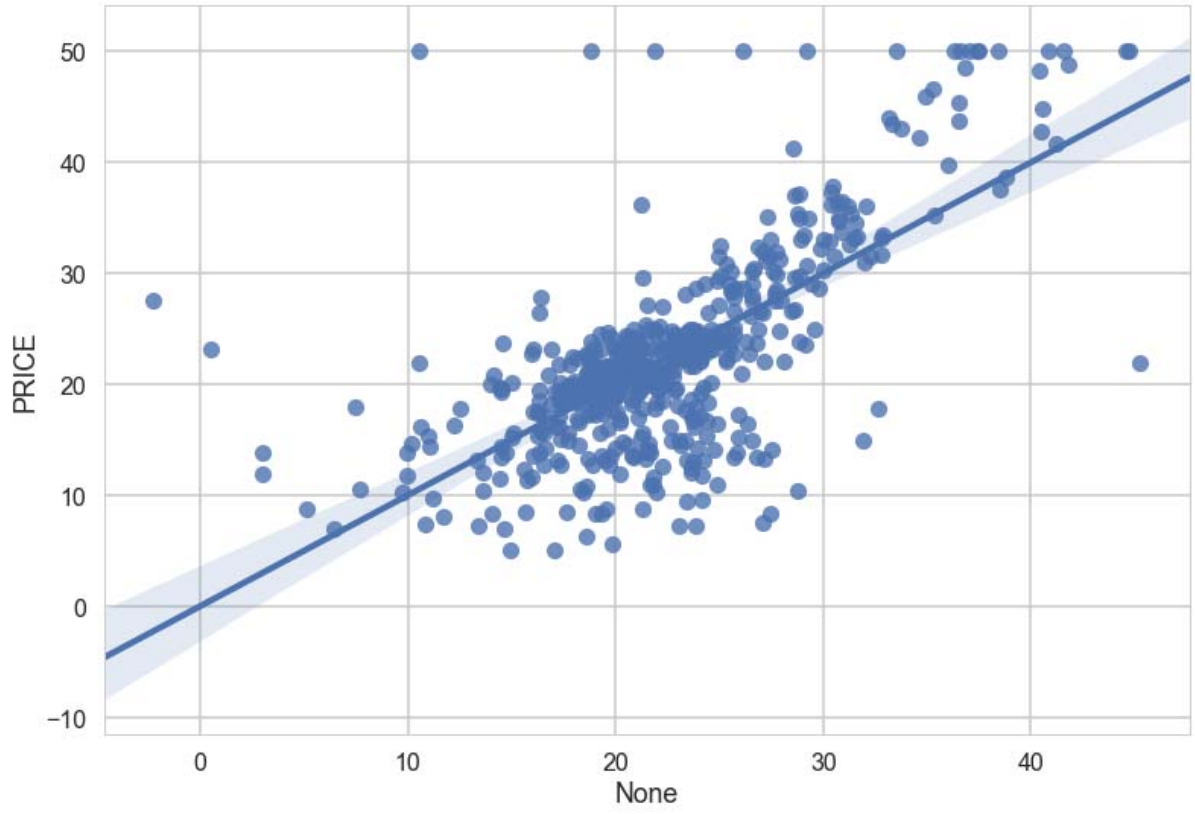
This is the very basic structure but it should be enough to get you through the homework. Things can get much more complex, for a quick run-down of further uses see the `statsmodels` help page (http://statsmodels.sourceforge.net/devel/example_formulas.html).

Let's see how our model actually fit our data. We can see below that there is a ceiling effect, we should probably look into that. Also, for large values of Y we get underpredictions, most predictions are below the 45-degree gridlines.

Your turn: Create a scatterpot between the predicted prices, available in `m.fittedvalues` and the original prices. How does the plot look?

In [23]:
```
# your turn
sns.regplot(x=m.fittedvalues, y=bos.PRICE)
```

Out[23]: <matplotlib.axes._subplots.AxesSubplot at 0xcf5ffd0>



Fitting Linear Regression using sklearn

In [24]:
```
from sklearn.linear_model import LinearRegression
X = bos.drop('PRICE', axis = 1)

# This creates a LinearRegression object
lm = LinearRegression()
lm
```

Out[24]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)

What can you do with a LinearRegression object?

Check out the scikit-learn docs here (http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html). We have listed the main functions here.

Main functions	Description
`lm.fit()`	Fit a linear model
`lm.predit()`	Predict Y using the linear model with estimated coefficients
`lm.score()`	Returns the coefficient of determination (R^2). *A measure of how well observed outcomes are replicated by the model, as the proportion of total variation of outcomes explained by the model*

What output can you get?

In [25]:
```
# Look inside lm object
#lm.residues_
```

Output	Description
`lm.coef_`	Estimated coefficients
`lm.intercept_`	Estimated intercept

Fit a linear model

The `lm.fit()` function estimates the coefficients the linear regression using least squares.

In [26]:
```
# Use all 13 predictors to fit linear regression model
lm.fit(X, bos.PRICE)
lm.coef_
```

Out[26]:
```
array([ -1.07170557e-01,   4.63952195e-02,   2.08602395e-02,
         2.68856140e+00,  -1.77957587e+01,   3.80475246e+00,
         7.51061703e-04,  -1.47575880e+00,   3.05655038e-01,
        -1.23293463e-02,  -9.53463555e-01,   9.39251272e-03,
        -5.25466633e-01])
```

Your turn: How would you change the model to not fit an intercept term? Would you recommend not having an intercept?

In [27]:
```
lm = LinearRegression(fit_intercept=False)
lm.fit(X, bos.PRICE)

lm = LinearRegression()
lm.fit(X, bos.PRICE)
```

Out[27]:
```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)
```

Estimated intercept and coefficients

Let's look at the estimated coefficients from the linear model using `1m.intercept_` and `lm.coef_`.

After we have fit our linear regression model using the least squares method, we want to see what are the estimates of our coefficients β_0, β_1, ..., β_{13}:

$$\hat{\beta}_0, \hat{\beta}_1, \ldots, \hat{\beta}_{13}$$

In [28]:
```
print('Estimated intercept coefficient:', lm.intercept_)
```

```
Estimated intercept coefficient: 36.4911032804
```

In [29]:
```
print('Number of coefficients:', len(lm.coef_))
```

```
Number of coefficients: 13
```

In [31]:
```
# The coefficients
pd.DataFrame(zip(X.columns, lm.coef_), columns = ['features', 'estimatedCoeffi
cients'])
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-31-fbaaa6358bf9> in <module>()
      1 # The coefficients
----> 2 pd.DataFrame(zip(X.columns, lm.coef_), columns = ['features', 'estima
tedCoefficients'])

C:\Users\amungale\AppData\Local\Continuum\Anaconda3\lib\site-packages\pandas
\core\frame.py in __init__(self, data, index, columns, dtype, copy)
    323                 mgr = self._init_dict({}, index, columns,
dtype=dtype)
    324         elif isinstance(data, collections.Iterator):
--> 325             raise TypeError("data argument can't be an iterator")
    326         else:
    327             try:

TypeError: data argument can't be an iterator
```

Predict Prices

We can calculate the predicted prices ($\hat{Y}_i$) using `lm.predict`.

$$\hat{Y}_i = \hat{\beta}_0 + \hat{\beta}_1 X_1 + \ldots \hat{\beta}_{13} X_{13}$$

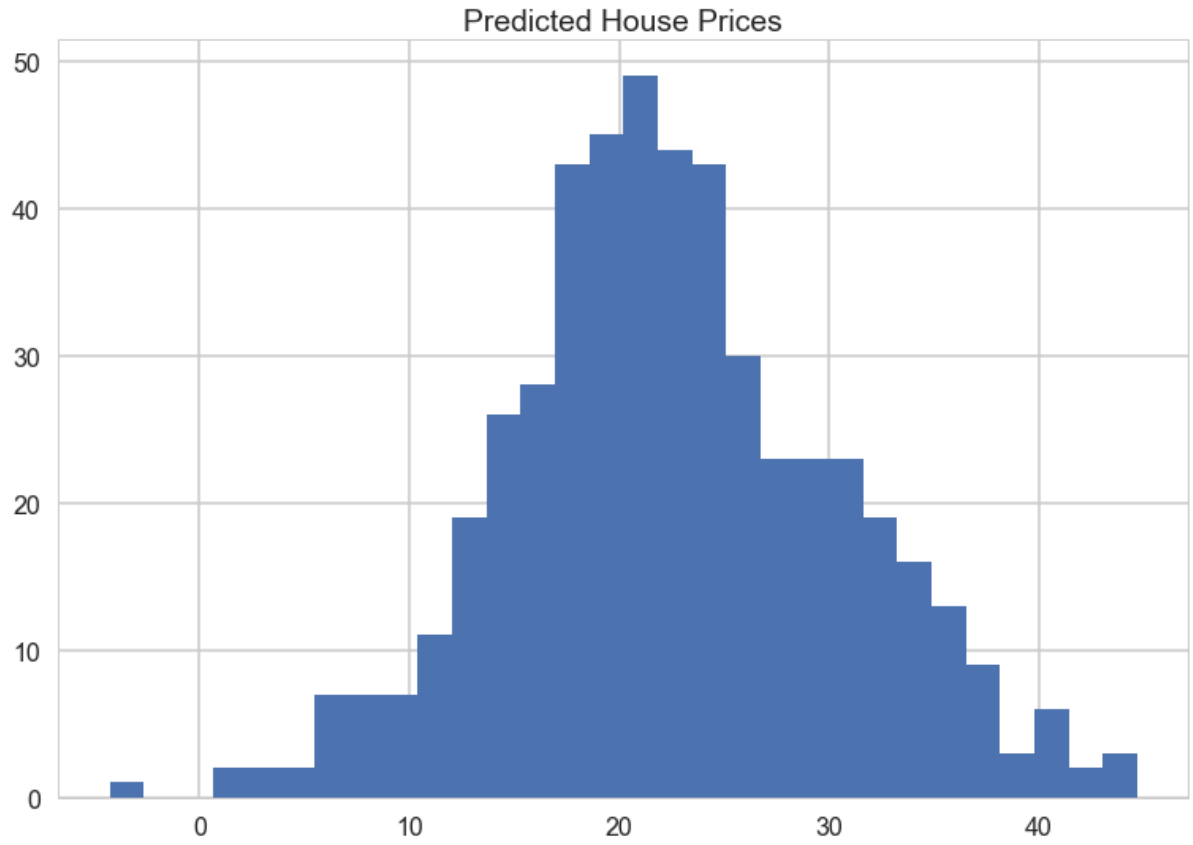
In [32]:
```
# first five predicted prices
lm.predict(X)[0:5]
```

Out[32]:
```
array([ 30.00821269,  25.0298606 ,  30.5702317 ,  28.60814055,  27.94288232])
```

Your turn:

- Histogram: Plot a histogram of all the predicted prices
- Scatter Plot: Let's plot the true prices compared to the predicted prices to see they disagree (we did this with `statsmodels` before).

In [33]:
```
# your turn
plt.hist(lm.predict(X), bins=30)
plt.title('Predicted House Prices')
plt.show()
```



Residual sum of squares

Let's calculate the residual sum of squares

$$S = \sum_{i=1}^{N} r_i = \sum_{i=1}^{N} (y_i - (\beta_0 + \beta_1 x_i))^2$$

In [34]:
```
print(np.sum((bos.PRICE - lm.predict(X)) ** 2))
```

```
11080.276284149868
```

Mean squared error

This is simple the mean of the residual sum of squares.

Your turn: Calculate the mean squared error and print it.

In [35]:
```
#your turn
np.sum((bos.PRICE - lm.predict(X)) ** 2)/len(bos)
```

Out[35]: 21.897779217687486

Relationship between PTRATIO and housing price

Try fitting a linear regression model using only the 'PTRATIO' (pupil-teacher ratio by town)

Calculate the mean squared error.

In [36]:
```
lm = LinearRegression()
lm.fit(X[['PTRATIO']], bos.PRICE)
```

Out[36]: LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normalize=False)

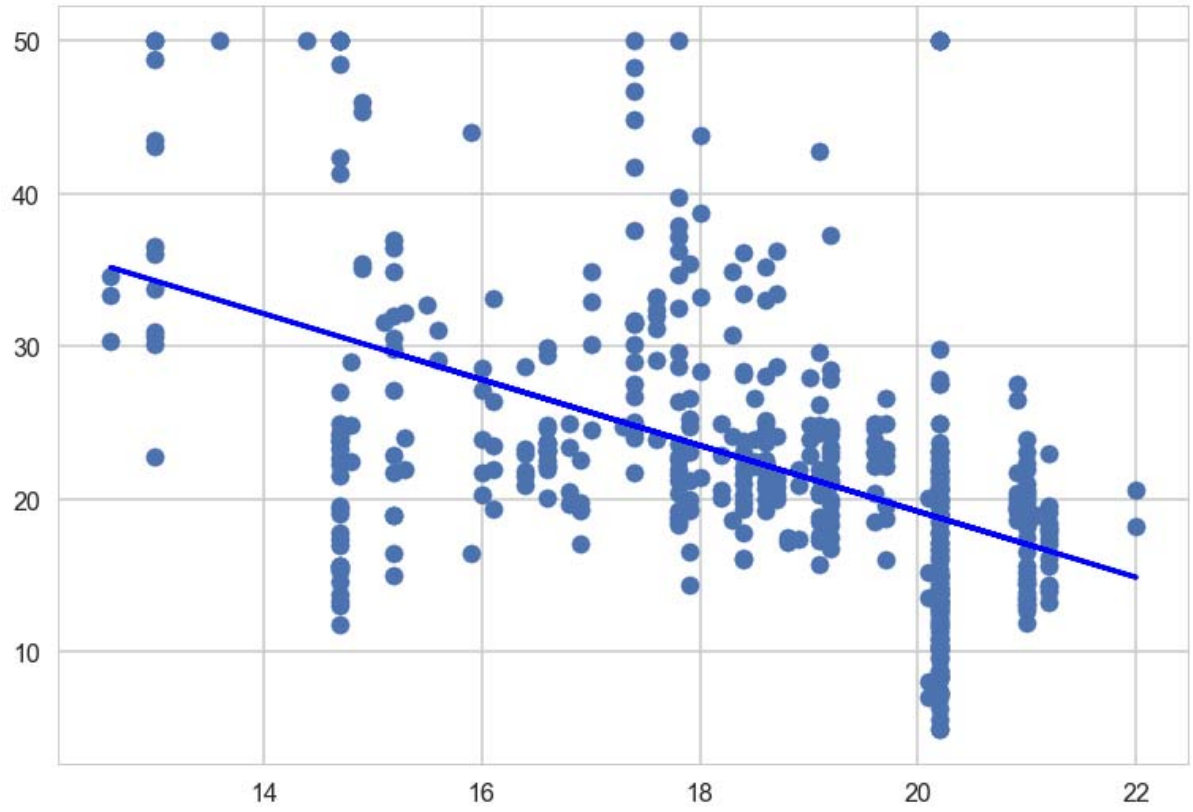
In [37]:
```
msePTRATIO = np.mean((bos.PRICE - lm.predict(X[['PTRATIO']])) ** 2)
print(msePTRATIO)
```

```
62.65220001376927
```

We can also plot the fitted linear regression line.

In [38]:
```
plt.scatter(bos.PTRATIO, bos.PRICE)
#plt.xlabel("Pupil-to-Teacher Ratio (PTRATIO)")
#plt.ylabel("Housing Price")
#plt.title("Relationship between PTRATIO and Price")

plt.plot(bos.PTRATIO, lm.predict(X[['PTRATIO']]), color='blue', linewidth=3)
plt.show()
```



Your turn

Try fitting a linear regression model using three independent variables

1. 'CRIM' (per capita crime rate by town)
2. 'RM' (average number of rooms per dwelling)
3. 'PTRATIO' (pupil-teacher ratio by town)

Calculate the mean squared error.

In [39]:
```
# your turn
lm.fit(X[['CRIM', 'RM', 'PTRATIO']], bos.PRICE)
mse = np.mean((bos.PRICE - lm.predict(X[['CRIM', 'RM', 'PTRATIO']])) ** 2)
mse
```

Out[39]:
```
34.32379656468118
```

Other important things to think about when fitting a linear regression model

- **Linearity**. The dependent variable Y is a linear combination of the regression coefficients and the independent variables X.
- **Constant standard deviation**. The SD of the dependent variable Y should be constant for different values of X.
  - e.g. PTRATIO
- **Normal distribution for errors**. The ϵ term we discussed at the beginning are assumed to be normally distributed.
  $$\epsilon_i \sim N(0, \sigma^2)$$
  Sometimes the distributions of responses Y may not be normally distributed at any given value of X. e.g. skewed positively or negatively.
- **Independent errors**. The observations are assumed to be obtained independently.
  - e.g. Observations across time may be correlated

Part 3: Training and Test Data sets

Purpose of splitting data into Training/testing sets

Let's stick to the linear regression example:

- We built our model with the requirement that the model fit the data well.
- As a side-effect, the model will fit **THIS** dataset well. What about new data?
  - We wanted the model for predictions, right?
- One simple solution, leave out some data (for **testing**) and **train** the model on the rest
- This also leads directly to the idea of cross-validation, next section.

One way of doing this is you can create training and testing data sets manually.

In [40]:
```
X_train = X[:-50]
X_test = X[-50:]
Y_train = bos.PRICE[:-50]
Y_test = bos.PRICE[-50:]
```

Another way, is to split the data into random train and test subsets using the function `train_test_split` in `sklearn.cross_validation`. Here's the documentation (http://scikit-learn.org/stable/modules/generated/sklearn.cross_validation.train_test_split.html).

In [43]:
```
X_train, X_test, Y_train, Y_test = sklearn.model_selection.train_test_split(
    X, bos.PRICE, test_size=0.33, random_state = 5)
```

Your turn: Let's build a linear regression model using our new training data sets.

- Fit a linear regression model to the training set
- Predict the output on the test set

In [48]:
```
# your turn
lm.fit(X_train, Y_train)
lm.predict(X_test)
```

Out[48]:
```
array([ 37.46723562,  31.39154701,  27.1201962 ,   6.46843347,
        33.62966737,   5.67067989,  27.03946671,  29.92704748,
        26.35661334,  22.45246021,  32.20504441,  21.78641653,
        23.41138441,  33.60894362,  28.28619511,  15.13859055,
         0.30087325,  18.71850376,  14.4706712 ,  11.10823598,
         2.69494197,  19.21693734,  38.41159345,  24.36936442,
        31.61493439,  11.42210397,  24.92862188,  23.31178043,
        22.7764079 ,  20.65081211,  16.035198  ,   7.07978633,
        17.65509209,  22.81470561,  29.21943405,  18.61354566,
        28.37701843,   8.80516873,  41.65140459,  34.02910176,
        20.1868926 ,   3.95600857,  29.69124564,  12.18081256,
        27.19403498,  30.63699231,  -6.24952457,  19.9462404 ,
        21.55123979,  13.36478173,  20.39068171,  19.87353324,
        23.57656877,  13.40141285,  17.66457201,  24.77424747,
        35.31476509,  15.48318159,  28.50764575,  21.72575404,
        20.58142839,  26.08460856,  14.51816968,  32.37494056,
        20.80917392,  12.18932524,  19.45551285,  25.23390429,
        21.77302317,  21.30227044,  20.58222113,  26.74635016,
        17.53006166,  18.7191946 ,  19.03026793,  25.76553031,
        21.8757557 ,  15.70891861,  35.12411848,  18.04488652,
        22.43612549,  39.4000555 ,  22.30677551,  14.9738331 ,
        25.29300631,  17.3200635 ,  18.58435124,  10.01693133,
        19.62408198,  17.24471407,  36.26263664,  17.55591517,
        21.10848471,  19.08435242,  24.72519887,  28.0878012 ,
        12.25474746,  22.40592558,  21.00483315,  13.51073355,
        23.09169468,  21.48906423,  14.14959117,  42.75677509,
         2.01088993,  21.9914102 ,  18.32505073,  22.59335404,
        28.93052931,  18.49024451,  27.61537531,  24.65547955,
        20.32508475,  32.66905896,  19.72975821,  12.8254    ,
        22.68957624,  18.2350211 ,  19.40432885,  16.19144346,
        21.77804736,  35.50387944,  22.24038654,  20.20025029,
        24.54270446,  25.29795497,  20.50220669,  23.0150761 ,
        23.38446711,  40.91809141,  37.84906745,  27.54024335,
        12.53470565,  15.90588084,  18.25352202,  21.62847325,
        15.77967465,   5.62636735,  24.00046271,  30.37118947,
        23.01126707,  18.29104509,  16.194709  ,  21.60846672,
        34.71665914,  23.40506116,  30.13747943,  18.0951727 ,
        22.16844264,  29.0922559 ,  13.36146671,  31.8608905 ,
        13.1643678 ,  13.91761543,  26.52314446,  31.39481197,
        10.62913801,  24.6869924 ,  28.95650935,  32.31758322,
        15.87113569,  29.94335724,   9.71836876,  34.70520017,
        25.70410195,  20.15430904,  15.3946584 ])
```

Your turn:

Calculate the mean squared error

- using just the test data
- using just the training data

Are they pretty similar or very different? What does that mean?

In [52]:
```
# your turn
mse_train = np.mean((Y_train - lm.predict(X_train)) ** 2)
mse_test = np.mean((Y_test - lm.predict(X_test)) ** 2)
mse_train, mse_test
```

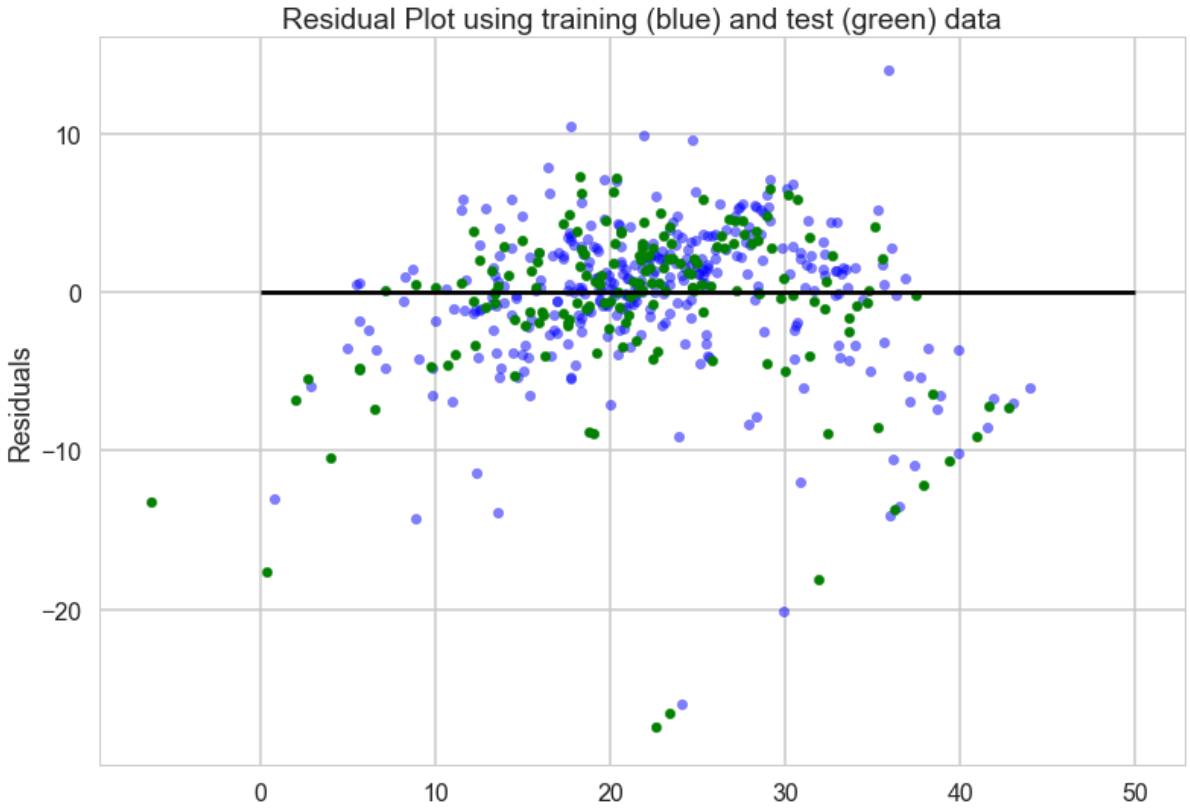
Out[52]: (19.54675847353466, 28.541367275619013)

As we would expect, the MSE test is higher than the MSE train since we built our predictive model using the train data.

Residual plots

In [53]:
```
plt.scatter(lm.predict(X_train), lm.predict(X_train) - Y_train, c='b', s=40, a
lpha=0.5)
plt.scatter(lm.predict(X_test), lm.predict(X_test) - Y_test, c='g', s=40)
plt.hlines(y = 0, xmin=0, xmax = 50)
plt.title('Residual Plot using training (blue) and test (green) data')
plt.ylabel('Residuals')
```

Out[53]: <matplotlib.text.Text at 0xe3cf278>



Your turn: Do you think this linear regression model generalizes well on the test data?

Yes. The random scatter of the residuals in green indicates that the linear model is a predictor of the test data.

K-fold Cross-validation as an extension of this idea

A simple extension of the Test/train split is called K-fold cross-validation.

Here's the procedure:

- randomly assign your n samples to one of K groups. They'll each have about n/k samples
- For each group k:
  - Fit the model (e.g. run regression) on all data excluding the k^{th} group
  - Use the model to predict the outcomes in group k
  - Calculate your prediction error for each observation in k^{th} group (e.g. $(Y_i - \hat{Y}_i)^2$ for regression, $1(Y_i = \hat{Y}_i)$ for logistic regression).
- Calculate the average prediction error across all samples $Err_{CV} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$

Luckily you don't have to do this entire process all by hand (`for` loops, etc.) every single time, `sci-kit learn` has a very nice implementation of this, have a look at the documentation (http://scikit-learn.org/stable/modules/cross_validation.html).

Your turn (extra credit): Implement K-Fold cross-validation using the procedure above and Boston Housing data set using $K = 4$. How does the average prediction error compare to the train-test split above?

In [68]:
```
from sklearn.model_selection import KFold, cross_val_score
kf = KFold(n_splits=4)
```

In [78]:
```
scores = cross_val_score(lm, X, bos['PRICE'],
 scoring='neg_mean_squared_error', cv=kf, n_jobs=1)
abs(scores)
```

Out[78]:
```
array([ 11.69977729,  39.0720141 ,  57.59307099,  61.59301573])
```