Creating a regression model in Python

**What is the problem we want to solve?**

We want to create an estimate of the linear relationship between variables, print the coefficients of correlation, and plot a line of best fit. For this analysis, I’ll be using data from the [House Sales in King’s County data set](https://www.kaggle.com/harlfoxem/housesalesprediction/kernels) from Kaggle. The King’s County data has information on house prices and house characteristics – so let’s see if we can estimate the relationship between house price and the square footage of the house.

**First step: Have the right data mining tools for the job – install Jupyter, and get familiar with a few modules.**

First things first, if you want to follow along, install Jupyter on your desktop. It’s a free platform that provides what is essentially a processer for iPython notebooks (.ipynb files) that is extremely intuitive to use.  We will be using the [Pandas](http://pandas.pydata.org/pandas-docs/stable/) module of Python to clean and restructure our data. Pandas is an open-source module for working with data structures and analysis, one that is ubiquitous for data scientists who use Python. It allows for data scientists to upload data in any format, and provides a simple platform organize, sort, and manipulate that data.

In [1]:

import pandas as pd

import matplotlib.pyplot as plt

import numpy as np

import scipy.stats as stats

import seaborn as sns

from matplotlib import rcParams

%matplotlib inline

%pylab inline

1. [**Numpy**](http://www.numpy.org/) – a necessary package for scientific computation. It includes an incredibly versatile structure for working with arrays, which are the primary data format that scikit-learn uses for input data.
2. [**Matplotlib**](http://matplotlib.org/) – the fundamental package for data visualization in Python. This module allows for the creation of everything from simple scatter plots to 3-dimensional contour plots. Note that from matplotlib we install pyplot, which is the highest order state-machine environment in the modules hierarchy (if that is meaningless to you don’t worry about it, just make sure you get it imported to your notebook). Using ‘%matplotlib inline’ is essential to make sure that all plots show up in your notebook.
3. [**Scipy**](https://www.scipy.org/) – a collection of tools for statistics in python. Stats is the scipy module that imports regression analysis functions.

In [2]:

df = pd.read\_csv('/Users/Desktop/kc\_house\_data.csv')

df.head()

Out[2]:

|  | **id** | **date** | **price** | **bedrooms** | **bathrooms** | **sqft\_living** | **sqft\_lot** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 7129300520 | 20141013T000000 | 221900.0 | 3 | 1.00 | 1180 | 5650 |
| **1** | 6414100192 | 20141209T000000 | 538000.0 | 3 | 2.25 | 2570 | 7242 |
| **2** | 5631500400 | 20150225T000000 | 180000.0 | 2 | 1.00 | 770 | 10000 |
| **3** | 2487200875 | 20141209T000000 | 604000.0 | 4 | 3.00 | 1960 | 5000 |
| **4** | 1954400510 | 20150218T000000 | 510000.0 | 3 | 2.00 | 1680 | 8080 |

In [3]:

df.isnull().any()

Out[3]:

id False

date False

price False

bedrooms False

bathrooms False

sqft\_living False

sqft\_lot False

...

dtype: bool

*Checking to see if any of our data has null values. If there were any, we’d drop or filter the null values out.*

In [4]:

df.dtypes

Out[4]:

id int64

date object

price float64

bedrooms int64

bathrooms float64

sqft\_living int64

sqft\_lot int64

...

dtype: object

*Checking out the data types for each of our variables. We want to get a sense of whether or not data is numerical (int64, float64) or not (object).*

I imported the data frame from the csv file using Pandas, and the first thing I did was make sure it reads properly. I also used the “isnull()” function to make sure that none of my data is unusable for regression. In real life, a single column may have data in the form of integers, strings, or NaN, all in one place – meaning that you need to check to make sure the types are matching and are suitable for regression. This data set happens to have been very rigorously prepared, something you won’t see often in your own database.

**Next: Simple exploratory analysis and regression results.**

Let’s get an understanding of the data before we go any further, it’s important to look at the shape of the data – and to double check if the data is reasonable. Corrupted data is not uncommon so it’s good practice to always run two checks: first, use df.describe() to look at all the variables in your analysis. Second, plot histograms of the variables that the analysis is targeting using plt.pyplot.hist().

In [5]:

df.describe()

Out[5]:

|  | **price** | **bedrooms** | **bathrooms** | **sqft\_living** |
| --- | --- | --- | --- | --- |
| **count** | 21613 | 21613 | 21613 | 21613 |
| **mean** | 540088.10 | 3.37 | 2.11 | 2079.90 |
| **std** | 367127.20 | 0.93 | 0.77 | 918.44 |
| **min** | 75000.00 | 0.00 | 0.00 | 290.00 |
| **25%** | 321950.00 | 3.00 | 1.75 | 1427.00 |
| **50%** | 450000.00 | 3.00 | 2.25 | 1910.00 |
| **75%** | 645000.00 | 4.00 | 2.50 | 2550.00 |
| **max** | 7700000.00 | 33.00 | 8.00 | 13540.00 |

*Quick takeaways: We are working with a data set that contains 21,613 observations, mean price is approximately $540k, median price is approximately $450k, and the average house’s area is 2080 ft2*

In [19]:

fig = plt.figure(figsize=(12, 6))

sqft = fig.add\_subplot(121)

cost = fig.add\_subplot(122)

sqft.hist(df.sqft\_living, bins=80)

sqft.set\_xlabel('Ft^2')

sqft.set\_title("Histogram of House Square Footage")

cost.hist(df.price, bins=80)

cost.set\_xlabel('Price ($)')

cost.set\_title("Histogram of Housing Prices")

plt.show()

*Using matplotlib (plt) we printed two histograms to observe the distribution of housing prices and square footage. What we find is that both variables have a distribution that is right-skewed.*  
  
Now that we have a good sense of our data set and know the distributions of the variables we are trying to measure, let’s do some regression analysis. First we import statsmodels to get the least squares regression estimator function. The [“Ordinary Least Squares”](https://en.wikipedia.org/wiki/Ordinary_least_squares) module will be doing the bulk of the work when it comes to crunching numbers for regression in Python.

In [15]:

import statsmodels.api as sm

from statsmodels.formula.api import ols

When you code to produce a linear regression summary with OLS with only two variables this will be the formula that you use:

*Reg = ols(‘Dependent variable ~ independent variable(s), dataframe).fit()*

*print(Reg.summary())*

When we look at housing prices and square footage for houses in King’s county, we print out the following summary report:

In [16]:

m = ols('price ~ sqft\_living',df).fit()

print (m.summary())

OLS Regression Results

==============================================================================

Dep. Variable: price R-squared: 0.493

Model: OLS Adj. R-squared: 0.493

Method: Least Squares F-statistic: 2.100e+04

Date: Fri, 23 Sep 2016 Prob (F-statistic): 0.00

Time: 14:46:10 Log-Likelihood: -3.0027e+05

No. Observations: 21613 AIC: 6.005e+05

Df Residuals: 21611 BIC: 6.006e+05

Df Model: 1

Covariance Type: nonrobust

===============================================================================

coef std err t P>|t| [95.0% Conf. Int.]

-------------------------------------------------------------------------------

Intercept -4.358e+04 4402.690 -9.899 0.000 -5.22e+04 -3.5e+04

sqft\_living 280.6236 1.936 144.920 0.000 276.828 284.419

==============================================================================

Omnibus: 14832.490 Durbin-Watson: 1.983

Prob(Omnibus): 0.000 Jarque-Bera (JB): 546444.713

Skew: 2.824 Prob(JB): 0.00

Kurtosis: 26.977 Cond. No. 5.63e+03

==============================================================================

Warnings:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The condition number is large, 5.63e+03. This might indicate that there are

strong multicollinearity or other numerical problems.

*An example of a simple linear regression model summary output.*

When you print the summary of the OLS regression, all relevant information can be easily found, including R-squared, t-statistics, standard error, and the coefficients of correlation. Looking at the output, it’s clear that there is an extremely significant relationship between square footage and housing prices since there is an extremely high t-value of 144.920, and a P>|t| of 0%–which essentially means that this relationship has a near-zero chance of being due to statistical variation or chance.

This relationship also has a decent magnitude – for every additional 100 square-feet a house has, we can predict that house to be priced $28,000 dollars higher on average. It is easy to adjust this formula to include more than one independent variable, simply follow the formula:

*Reg = ols(‘Dependent variable ~ivar1 + ivar2 + ivar3… + ivarN, dataframe).fit()*

*print(Reg.summary())*

In [26]:

m = ols('price ~ sqft\_living + bedrooms + grade + condition',df).fit()

print (m.summary())

OLS Regression Results

==============================================================================

Dep. Variable: price R-squared: 0.555

Model: OLS Adj. R-squared: 0.555

Method: Least Squares F-statistic: 6749.

Date: Fri, 23 Sep 2016 Prob (F-statistic): 0.00

Time: 15:11:41 Log-Likelihood: -2.9884e+05

No. Observations: 21613 AIC: 5.977e+05

Df Residuals: 21608 BIC: 5.977e+05

Df Model: 4

Covariance Type: nonrobust

===============================================================================

coef std err t P>|t| [95.0% Conf. Int.]

-------------------------------------------------------------------------------

Intercept -7.398e+05 1.81e+04 -40.855 0.000 -7.75e+05 -7.04e+05

sqft\_living 212.3034 3.249 65.353 0.000 205.936 218.671

bedrooms -4.568e+04 2222.205 -20.555 0.000 -5e+04 -4.13e+04

grade 1.001e+05 2241.553 44.673 0.000 9.57e+04 1.05e+05

condition 6.615e+04 2598.352 25.457 0.000 6.11e+04 7.12e+04

==============================================================================

Omnibus: 16773.778 Durbin-Watson: 1.988

Prob(Omnibus): 0.000 Jarque-Bera (JB): 973426.793

Skew: 3.249 Prob(JB): 0.00

Kurtosis: 35.229 Cond. No. 2.50e+04

==============================================================================

Warnings:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The condition number is large, 2.5e+04. This might indicate that there are

strong multicollinearity or other numerical problems.

*An example of multivariate linear regression.*

In our multivariate regression output above, we learn that by using additional independent variables, such as the number of bedrooms, we can provide a model that fits the data better, as the R-squared for this regression has increased to 0.555. This means that we went from being able to explain about 49.3% of the variation in the model to 55.5% with the addition of a few more independent variables.

**Visualizing the regression results.**

Having the regression summary output is important for checking the accuracy of the regression model and data to be used for estimation and prediction – but visualizing the regression is an important step to take to communicate the results of the regression in a more digestible format.

This section will rely entirely on Seaborn (sns), which has an incredibly simple and intuitive function for graphing regression lines with scatterplots. I chose to create a jointplot for square footage and price that shows the regression line as well as distribution plots for each variable.

In [24]:

sns.jointplot(x="sqft\_living", y="price", data=df, kind = 'reg',fit\_reg= True, size = 7)

plt.show()

/Users/michaelrundell/anaconda/lib/python3.5/site-packages/statsmodels/nonparametric/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number instead of an integer will result in an error in the future

y = X[:m/2+1] + np.r\_[0,X[m/2+1:],0]\*1j

That wraps up my regression example, but there are many other ways to perform regression analysis in python, especially when it comes to using certain techniques. For more on regression models, consult the resources below. Next, we’ll cover cluster analysis.

* [**Visualizing linear relationships using Seaborn**](https://stanford.edu/~mwaskom/software/seaborn/tutorial/regression.html)– this documentation gives specific examples that show how to modify you regression plots, and display new features that you might not know how to code yourself. It also teaches you how to fit different kinds of models, such as quadratic or logistic models.  
    
  [**Statistics in Python**](http://www.scipy-lectures.org/packages/statistics/index.html) – this tutorial covers different techniques for performing regression in python, and also will teach you how to do hypothesis testing and testing for interactions.

If you want to learn about more data mining software that helps you with visualizing your results, you should look at these [31 free data visualization tools](https://www.springboard.com/blog/31-free-data-visualization-tools/) we’ve compiled.

Creating a Clustering Model in Python

**What is the problem we want to solve?**

We want to create natural groupings for a set of data objects that might not be explicitly stated in the data itself. Our analysis will use data on the eruptions from Old Faithful, the famous geyser in Yellowstone Park. The data is found from [this Github repository](https://github.com/barneygovan/from-data-with-love) by Barney Govan. It contains only two attributes, waiting time between eruptions (minutes) and length of eruption (minutes). Having only two attributes makes it easy to create a simple k-means cluster model.

**What is a k-means cluster model?**

K-Means Cluster models work in the following way – all credit to this blog:

1. Start with a randomly selected set of k centroids (the supposed centers of the k clusters)
2. Determine which observation is in which cluster, based on which centroid it is closest to (using the squared Euclidean distance: ∑pj=1(xij−xi′j)2 where p is the number of dimensions.
3. Recalculate the centroids of each cluster by minimizing the squared Euclidean distance to each observation in the cluster
4. Repeat 2. and 3. until the members of the clusters (and hence the positions of the centroids) no longer change.

If this is still confusing, check out [this helpful video](https://www.youtube.com/watch?v=aiJ8II94qck)by Jigsaw Academy. For now, let’s move on to applying this technique to our Old Faithful data set.

**Step One: Exploratory Data Analysis**

You will need to install a few modules, including one new module called **[Sci-kit Learn](http://scikit-learn.org/stable/)** – a collection of tools for machine learning and data mining in Python (read our tutorial on using Sci-kit for [Neural Network Models](https://www.springboard.com/blog/beginners-guide-neural-network-in-python-scikit-learn-0-18/)). Cluster is the sci-kit module that imports functions with clustering algorithms, hence why it is imported from sci-kit.

First, let’s import all necessary modules into our iPython Notebook and do some [exploratory data analysis](https://en.wikipedia.org/wiki/Exploratory_data_analysis).

In [18]:

import pandas as pd

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import sklearn

from sklearn import cluster

%matplotlib inline

faithful = pd.read\_csv('/Users/michaelrundell/Desktop/faithful.csv')

faithful.head()

Out[18]:

|  | **eruptions** | **waiting** |
| --- | --- | --- |
| **0** | 3.600 | 79 |
| **1** | 1.800 | 54 |
| **2** | 3.333 | 74 |
| **3** | 2.283 | 62 |
| **4** | 4.533 | 85 |

*Reading the old faithful csv and importing all necessary values*

All I’ve done is read the csv from my local directory, which happens to be my computer’s desktop, and shown the first 5 entries of the data. Fortunately, I know this data set has no columns with missing or NaN values, so we can skip the data cleaning section in this example. Let’s take a look at a basic scatterplot of the data.

In [19]:

faithful.columns = ['eruptions', 'waiting']

plt.scatter(faithful.eruptions, faithful.waiting)

plt.title('Old Faithful Data Scatterplot')

plt.xlabel('Length of eruption (minutes)')

plt.ylabel('Time between eruptions (minutes)')

Out[19]:

<matplotlib.text.Text at 0x12a29bba8>

*Renaming the columns and using matplotlib to create a simple scatterplot.*

Some quick notes on my process here: I renamed the columns – they don’t look any different to the naked eye, but the “waiting” column had an extra space before the word, and to prevent any confusion with further analysis I changed it to ensure I don’t forget or make any mistakes down the road.

**Step two: Building the cluster model**

What we see is a scatter plot that has two clusters that are easily apparent, but the data set does not label any observation as belonging to either group. The next few steps will cover the process of visually differentiating the two groups. In the code below, I establish some important variables and alter the format of the data.

In [20]:

faith = np.array(faithful)

k = 2

kmeans = cluster.KMeans(n\_clusters=k)

kmeans.fit(faith)

labels = kmeans.labels\_

centroids = kmeans.cluster\_centers\_

*Formatting and function creation.*

1. I read the faithful dataframe as a numpy array in order for sci-kit to be able to read the data.
2. K = 2 was chosen as the number of clusters because there are 2 clear groupings we are trying to create.
3. The ‘kmeans’ variable is defined by the output called from the cluster module in sci-kit. We have it take on a K number of clusters, and fit the data in the array ‘faith’.

Now that we have set up the variables for creating a cluster model, let’s create a visualization. The code below will plot a scatter plot that colors by cluster, and gives final centroid locations. Explanation of specific lines of code can be found below.

In [21]:

for i in range(k):

# select only data observations with cluster label == i

ds = faith[np.where(labels==i)]

# plot the data observations

plt.plot(ds[:,0],ds[:,1],'o', markersize=7)

# plot the centroids

lines = plt.plot(centroids[i,0],centroids[i,1],'kx')

# make the centroid x's bigger

plt.setp(lines,ms=15.0)

plt.setp(lines,mew=4.0)

plt.show()

*Creating a visualization of the cluster model.*

A quick breakdown of the code above:

1. All of the work done to group the data into 2 groups was done in the previous section of code where we used the command kmeans.fit(faith). This section of the code simply creates the plot that shows it.
2. The ds variable is simply the original data, but reformatted to include the new color labels based on the number of groups – the number of integers in k.
3. plt.plot calls the x-data, the y-data, the shape of the objects, and the size of the circles.
4. The rest of the code displays the final centroids of the k-means clustering process, and controls the size and thickness of the centroid markers.

And here we have it – a simple cluster model. This code can be adapted to include a different number of clusters, but for this problem it makes sense to include only two clusters. Now that we have these clusters that seem to be well defined, we can infer meaning from these two clusters. What do they stand for? The green cluster: consisting of mostly short eruptions with a brief waiting time between eruptions could be defined as ‘weak or rapid-fire’, while the blue cluster could be called ‘power’ eruptions.

Of note: this technique is not adaptable for all data sets –  data scientist David Robinson [explains it perfectly in his article](http://varianceexplained.org/r/kmeans-free-lunch/) that K-means clustering is “not a free lunch.” K-means has assumptions that fail if your data has uneven cluster probabilities (they don’t have approximately the same amount of observations in each cluster), or has non-spherical clusters. If you don’t think that your clustering problem will work well with K-means clustering, check out these resources on alternative cluster modeling techniques:

* [**Sci-kit Clustering Modules**](http://scikit-learn.org/stable/modules/clustering.html)**–**this documentation has a nifty image that visually compares the clustering algorithms in scikit-learn, as they look for different scatterplots. Using this documentation can point you to the right algorithm to use if you have a scatter plot similar to one of their examples. It also gives you some insight on how to evaluate your clustering model mathematically.
* [**Clustering Algorithms**](http://web.stanford.edu/class/cs345a/slides/12-clustering.pdf) – this Powerpoint presentation from Stanford’s CS345 course, Data Mining, gives insight into different techniques – how they work, where they are effective and ineffective, etc. It is a great learning resource to understand how clustering works at a theoretical level.

## Creating a regression model in Python

**What is the problem we want to solve?**

We want to create an estimate of the linear relationship between variables, print the coefficients of correlation, and plot a line of best fit. For this analysis, I’ll be using data from the [House Sales in King’s County data set](https://www.kaggle.com/harlfoxem/housesalesprediction/kernels) from Kaggle. If you’re unfamiliar with [Kaggle](https://www.kaggle.com/datasets?sortBy=votes&group=all), it’s a fantastic resource for finding data sets good for practicing data science. The King’s County data has information on house prices and house characteristics – so let’s see if we can estimate the relationship between house price and the square footage of the house.

**First step: Have the right data mining tools for the job – install Jupyter, and get familiar with a few modules.**

First things first, if you want to follow along, install Jupyter on your desktop. It’s a free platform that provides what is essentially a processer for iPython notebooks (.ipynb files) that is extremely intuitive to use. [Follow these instructions for installation](http://jupyter.readthedocs.io/en/latest/install.html). Everything I do here will be completed in a “Python [Root]” file in Jupyter.

We will be using the [Pandas](http://pandas.pydata.org/pandas-docs/stable/) module of Python to clean and restructure our data. Pandas is an open-source module for working with data structures and analysis, one that is ubiquitous for data scientists who use Python. It allows for data scientists to upload data in any format, and provides a simple platform organize, sort, and manipulate that data. If this is your first time using Pandas, check out [this awesome tutorial on the basic functions](http://nbviewer.jupyter.org/github/pybokeh/jupyter_notebooks/blob/master/pandas/PandasCheatSheet.ipynb)!

In [1]:

import pandas as pd

import matplotlib.pyplot as plt

import numpy as np

import scipy.stats as stats

import seaborn as sns

from matplotlib import rcParams

%matplotlib inline

%pylab inline

Populating the interactive namespace from numpy and matplotlib

In the code above I imported a few modules, here’s a breakdown of what they do:

1. [**Numpy**](http://www.numpy.org/) – a necessary package for scientific computation. It includes an incredibly versatile structure for working with arrays, which are the primary data format that scikit-learn uses for input data.
2. [**Matplotlib**](http://matplotlib.org/) – the fundamental package for data visualization in Python. This module allows for the creation of everything from simple scatter plots to 3-dimensional contour plots. Note that from matplotlib we install pyplot, which is the highest order state-machine environment in the modules hierarchy (if that is meaningless to you don’t worry about it, just make sure you get it imported to your notebook). Using ‘%matplotlib inline’ is essential to make sure that all plots show up in your notebook.
3. [**Scipy**](https://www.scipy.org/) – a collection of tools for statistics in python. Stats is the scipy module that imports regression analysis functions.

Let’s break down how to apply data mining to solve a regression problem step-by-step! In real life you most likely won’t be handed a dataset ready to have machine learning techniques applied right away, so you will need to clean and organize the data first.

In [2]:

df = pd.read\_csv('/Users/michaelrundell/Desktop/kc\_house\_data.csv')

df.head()

Out[2]:

|  | **id** | **date** | **price** | **bedrooms** | **bathrooms** | **sqft\_living** | **sqft\_lot** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 7129300520 | 20141013T000000 | 221900.0 | 3 | 1.00 | 1180 | 5650 |
| **1** | 6414100192 | 20141209T000000 | 538000.0 | 3 | 2.25 | 2570 | 7242 |
| **2** | 5631500400 | 20150225T000000 | 180000.0 | 2 | 1.00 | 770 | 10000 |
| **3** | 2487200875 | 20141209T000000 | 604000.0 | 4 | 3.00 | 1960 | 5000 |
| **4** | 1954400510 | 20150218T000000 | 510000.0 | 3 | 2.00 | 1680 | 8080 |

*Reading the csv file from Kaggle using pandas (pd.read\_csv).*

In [3]:

df.isnull().any()

Out[3]:

id False

date False

price False

bedrooms False

bathrooms False

sqft\_living False

sqft\_lot False

...

dtype: bool

*Checking to see if any of our data has null values. If there were any, we’d drop or filter the null values out.*

In [4]:

df.dtypes

Out[4]:

id int64

date object

price float64

bedrooms int64

bathrooms float64

sqft\_living int64

sqft\_lot int64

...

dtype: object

*Checking out the data types for each of our variables. We want to get a sense of whether or not data is numerical (int64, float64) or not (object).*

I imported the data frame from the csv file using Pandas, and the first thing I did was make sure it reads properly. I also used the “isnull()” function to make sure that none of my data is unusable for regression. In real life, a single column may have data in the form of integers, strings, or NaN, all in one place – meaning that you need to check to make sure the types are matching and are suitable for regression. This data set happens to have been very rigorously prepared, something you won’t see often in your own database.

**Next: Simple exploratory analysis and regression results.**

Let’s get an understanding of the data before we go any further, it’s important to look at the shape of the data – and to double check if the data is reasonable. Corrupted data is not uncommon so it’s good practice to always run two checks: first, use df.describe() to look at all the variables in your analysis. Second, plot histograms of the variables that the analysis is targeting using plt.pyplot.hist().

In [5]:

df.describe()

Out[5]:

|  | **price** | **bedrooms** | **bathrooms** | **sqft\_living** |
| --- | --- | --- | --- | --- |
| **count** | 21613 | 21613 | 21613 | 21613 |
| **mean** | 540088.10 | 3.37 | 2.11 | 2079.90 |
| **std** | 367127.20 | 0.93 | 0.77 | 918.44 |
| **min** | 75000.00 | 0.00 | 0.00 | 290.00 |
| **25%** | 321950.00 | 3.00 | 1.75 | 1427.00 |
| **50%** | 450000.00 | 3.00 | 2.25 | 1910.00 |
| **75%** | 645000.00 | 4.00 | 2.50 | 2550.00 |
| **max** | 7700000.00 | 33.00 | 8.00 | 13540.00 |

*Quick takeaways: We are working with a data set that contains 21,613 observations, mean price is approximately $540k, median price is approximately $450k, and the average house’s area is 2080 ft2*

In [19]:

fig = plt.figure(figsize=(12, 6))

sqft = fig.add\_subplot(121)

cost = fig.add\_subplot(122)

sqft.hist(df.sqft\_living, bins=80)

sqft.set\_xlabel('Ft^2')

sqft.set\_title("Histogram of House Square Footage")

cost.hist(df.price, bins=80)

cost.set\_xlabel('Price ($)')

cost.set\_title("Histogram of Housing Prices")

plt.show()

*Using matplotlib (plt) we printed two histograms to observe the distribution of housing prices and square footage. What we find is that both variables have a distribution that is right-skewed.*  
  
Now that we have a good sense of our data set and know the distributions of the variables we are trying to measure, let’s do some regression analysis. First we import statsmodels to get the least squares regression estimator function. The [“Ordinary Least Squares”](https://en.wikipedia.org/wiki/Ordinary_least_squares) module will be doing the bulk of the work when it comes to crunching numbers for regression in Python.

In [15]:

import statsmodels.api as sm

from statsmodels.formula.api import ols

When you code to produce a linear regression summary with OLS with only two variables this will be the formula that you use:

*Reg = ols(‘Dependent variable ~ independent variable(s), dataframe).fit()*

*print(Reg.summary())*

When we look at housing prices and square footage for houses in King’s county, we print out the following summary report:

In [16]:

m = ols('price ~ sqft\_living',df).fit()

print (m.summary())

OLS Regression Results

==============================================================================

Dep. Variable: price R-squared: 0.493

Model: OLS Adj. R-squared: 0.493

Method: Least Squares F-statistic: 2.100e+04

Date: Fri, 23 Sep 2016 Prob (F-statistic): 0.00

Time: 14:46:10 Log-Likelihood: -3.0027e+05

No. Observations: 21613 AIC: 6.005e+05

Df Residuals: 21611 BIC: 6.006e+05

Df Model: 1

Covariance Type: nonrobust

===============================================================================

coef std err t P>|t| [95.0% Conf. Int.]

-------------------------------------------------------------------------------

Intercept -4.358e+04 4402.690 -9.899 0.000 -5.22e+04 -3.5e+04

sqft\_living 280.6236 1.936 144.920 0.000 276.828 284.419

==============================================================================

Omnibus: 14832.490 Durbin-Watson: 1.983

Prob(Omnibus): 0.000 Jarque-Bera (JB): 546444.713

Skew: 2.824 Prob(JB): 0.00

Kurtosis: 26.977 Cond. No. 5.63e+03

==============================================================================

Warnings:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The condition number is large, 5.63e+03. This might indicate that there are

strong multicollinearity or other numerical problems.

An example of a simple linear regression model summary output.

When you print the summary of the OLS regression, all relevant information can be easily found, including R-squared, t-statistics, standard error, and the coefficients of correlation. Looking at the output, it’s clear that there is an extremely significant relationship between square footage and housing prices since there is an extremely high t-value of 144.920, and a P>|t| of 0%–which essentially means that this relationship has a near-zero chance of being due to statistical variation or chance.

This relationship also has a decent magnitude – for every additional 100 square-feet a house has, we can predict that house to be priced $28,000 dollars higher on average. It is easy to adjust this formula to include more than one independent variable, simply follow the formula:

*Reg = ols(‘Dependent variable ~ivar1 + ivar2 + ivar3… + ivarN, dataframe).fit()*

*print(Reg.summary())*

In [26]:

m = ols('price ~ sqft\_living + bedrooms + grade + condition',df).fit()

print (m.summary())

OLS Regression Results

==============================================================================

Dep. Variable: price R-squared: 0.555

Model: OLS Adj. R-squared: 0.555

Method: Least Squares F-statistic: 6749.

Date: Fri, 23 Sep 2016 Prob (F-statistic): 0.00

Time: 15:11:41 Log-Likelihood: -2.9884e+05

No. Observations: 21613 AIC: 5.977e+05

Df Residuals: 21608 BIC: 5.977e+05

Df Model: 4

Covariance Type: nonrobust

===============================================================================

coef std err t P>|t| [95.0% Conf. Int.]

-------------------------------------------------------------------------------

Intercept -7.398e+05 1.81e+04 -40.855 0.000 -7.75e+05 -7.04e+05

sqft\_living 212.3034 3.249 65.353 0.000 205.936 218.671

bedrooms -4.568e+04 2222.205 -20.555 0.000 -5e+04 -4.13e+04

grade 1.001e+05 2241.553 44.673 0.000 9.57e+04 1.05e+05

condition 6.615e+04 2598.352 25.457 0.000 6.11e+04 7.12e+04

==============================================================================

Omnibus: 16773.778 Durbin-Watson: 1.988

Prob(Omnibus): 0.000 Jarque-Bera (JB): 973426.793

Skew: 3.249 Prob(JB): 0.00

Kurtosis: 35.229 Cond. No. 2.50e+04

==============================================================================

Warnings:

[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

[2] The condition number is large, 2.5e+04. This might indicate that there are

strong multicollinearity or other numerical problems.

*An example of multivariate linear regression.*

In our multivariate regression output above, we learn that by using additional independent variables, such as the number of bedrooms, we can provide a model that fits the data better, as the R-squared for this regression has increased to 0.555. This means that we went from being able to explain about 49.3% of the variation in the model to 55.5% with the addition of a few more independent variables.

**Visualizing the regression results.**

Having the regression summary output is important for checking the accuracy of the regression model and data to be used for estimation and prediction – but visualizing the regression is an important step to take to communicate the results of the regression in a more digestible format.

This section will rely entirely on Seaborn (sns), which has an incredibly simple and intuitive function for graphing regression lines with scatterplots. I chose to create a jointplot for square footage and price that shows the regression line as well as distribution plots for each variable.

In [24]:

sns.jointplot(x="sqft\_living", y="price", data=df, kind = 'reg',fit\_reg= True, size = 7)

plt.show()

/Users/michaelrundell/anaconda/lib/python3.5/site-packages/statsmodels/nonparametric/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number instead of an integer will result in an error in the future

y = X[:m/2+1] + np.r\_[0,X[m/2+1:],0]\*1j

That wraps up my regression example, but there are many other ways to perform regression analysis in python, especially when it comes to using certain techniques. For more on regression models, consult the resources below. Next, we’ll cover cluster analysis.

* [**Visualizing linear relationships using Seaborn**](https://stanford.edu/~mwaskom/software/seaborn/tutorial/regression.html)– this documentation gives specific examples that show how to modify you regression plots, and display new features that you might not know how to code yourself. It also teaches you how to fit different kinds of models, such as quadratic or logistic models.  
    
  [**Statistics in Python**](http://www.scipy-lectures.org/packages/statistics/index.html) – this tutorial covers different techniques for performing regression in python, and also will teach you how to do hypothesis testing and testing for interactions.

If you want to learn about more data mining software that helps you with visualizing your results, you should look at these [31 free data visualization tools](https://www.springboard.com/blog/31-free-data-visualization-tools/) we’ve compiled.

## Creating a Clustering Model in Python

**What is the problem we want to solve?**

We want to create natural groupings for a set of data objects that might not be explicitly stated in the data itself. Our analysis will use data on the eruptions from Old Faithful, the famous geyser in Yellowstone Park. The data is found from [this Github repository](https://github.com/barneygovan/from-data-with-love) by Barney Govan. It contains only two attributes, waiting time between eruptions (minutes) and length of eruption (minutes). Having only two attributes makes it easy to create a simple k-means cluster model.

**What is a k-means cluster model?**

K-Means Cluster models work in the following way – all credit to this blog:

1. Start with a randomly selected set of k centroids (the supposed centers of the k clusters)
2. Determine which observation is in which cluster, based on which centroid it is closest to (using the squared Euclidean distance: ∑pj=1(xij−xi′j)2 where p is the number of dimensions.
3. Recalculate the centroids of each cluster by minimizing the squared Euclidean distance to each observation in the cluster
4. Repeat 2. and 3. until the members of the clusters (and hence the positions of the centroids) no longer change.

If this is still confusing, check out [this helpful video](https://www.youtube.com/watch?v=aiJ8II94qck)by Jigsaw Academy. For now, let’s move on to applying this technique to our Old Faithful data set.

**Step One: Exploratory Data Analysis**

You will need to install a few modules, including one new module called **[Sci-kit Learn](http://scikit-learn.org/stable/)** – a collection of tools for machine learning and data mining in Python (read our tutorial on using Sci-kit for [Neural Network Models](https://www.springboard.com/blog/beginners-guide-neural-network-in-python-scikit-learn-0-18/)). Cluster is the sci-kit module that imports functions with clustering algorithms, hence why it is imported from sci-kit.

First, let’s import all necessary modules into our iPython Notebook and do some [exploratory data analysis](https://en.wikipedia.org/wiki/Exploratory_data_analysis).

In [18]:

import pandas as pd

import numpy as np

import matplotlib

import matplotlib.pyplot as plt

import sklearn

from sklearn import cluster

%matplotlib inline

faithful = pd.read\_csv('/Users/michaelrundell/Desktop/faithful.csv')

faithful.head()

Out[18]:

|  | **eruptions** | **waiting** |
| --- | --- | --- |
| **0** | 3.600 | 79 |
| **1** | 1.800 | 54 |
| **2** | 3.333 | 74 |
| **3** | 2.283 | 62 |
| **4** | 4.533 | 85 |

*Reading the old faithful csv and importing all necessary values*

All I’ve done is read the csv from my local directory, which happens to be my computer’s desktop, and shown the first 5 entries of the data. Fortunately, I know this data set has no columns with missing or NaN values, so we can skip the data cleaning section in this example. Let’s take a look at a basic scatterplot of the data.

In [19]:

faithful.columns = ['eruptions', 'waiting']

plt.scatter(faithful.eruptions, faithful.waiting)

plt.title('Old Faithful Data Scatterplot')

plt.xlabel('Length of eruption (minutes)')

plt.ylabel('Time between eruptions (minutes)')

Out[19]:

<matplotlib.text.Text at 0x12a29bba8>

*Renaming the columns and using matplotlib to create a simple scatterplot.*

Some quick notes on my process here: I renamed the columns – they don’t look any different to the naked eye, but the “waiting” column had an extra space before the word, and to prevent any confusion with further analysis I changed it to ensure I don’t forget or make any mistakes down the road.

**Step two: Building the cluster model**

What we see is a scatter plot that has two clusters that are easily apparent, but the data set does not label any observation as belonging to either group. The next few steps will cover the process of visually differentiating the two groups. In the code below, I establish some important variables and alter the format of the data.

In [20]:

faith = np.array(faithful)

k = 2

kmeans = cluster.KMeans(n\_clusters=k)

kmeans.fit(faith)

labels = kmeans.labels\_

centroids = kmeans.cluster\_centers\_

*Formatting and function creation.*

1. I read the faithful dataframe as a numpy array in order for sci-kit to be able to read the data.
2. K = 2 was chosen as the number of clusters because there are 2 clear groupings we are trying to create.
3. The ‘kmeans’ variable is defined by the output called from the cluster module in sci-kit. We have it take on a K number of clusters, and fit the data in the array ‘faith’.

Now that we have set up the variables for creating a cluster model, let’s create a visualization. The code below will plot a scatter plot that colors by cluster, and gives final centroid locations. Explanation of specific lines of code can be found below.

In [21]:

for i in range(k):

# select only data observations with cluster label == i

ds = faith[np.where(labels==i)]

# plot the data observations

plt.plot(ds[:,0],ds[:,1],'o', markersize=7)

# plot the centroids

lines = plt.plot(centroids[i,0],centroids[i,1],'kx')

# make the centroid x's bigger

plt.setp(lines,ms=15.0)

plt.setp(lines,mew=4.0)

plt.show()

*Creating a visualization of the cluster model.*

A quick breakdown of the code above:

1. All of the work done to group the data into 2 groups was done in the previous section of code where we used the command kmeans.fit(faith). This section of the code simply creates the plot that shows it.
2. The ds variable is simply the original data, but reformatted to include the new color labels based on the number of groups – the number of integers in k.
3. plt.plot calls the x-data, the y-data, the shape of the objects, and the size of the circles.
4. The rest of the code displays the final centroids of the k-means clustering process, and controls the size and thickness of the centroid markers.

And here we have it – a simple cluster model. This code can be adapted to include a different number of clusters, but for this problem it makes sense to include only two clusters. Now that we have these clusters that seem to be well defined, we can infer meaning from these two clusters. What do they stand for? The green cluster: consisting of mostly short eruptions with a brief waiting time between eruptions could be defined as ‘weak or rapid-fire’, while the blue cluster could be called ‘power’ eruptions.

Of note: this technique is not adaptable for all data sets –  data scientist David Robinson [explains it perfectly in his article](http://varianceexplained.org/r/kmeans-free-lunch/) that K-means clustering is “not a free lunch.” K-means has assumptions that fail if your data has uneven cluster probabilities (they don’t have approximately the same amount of observations in each cluster), or has non-spherical clusters. If you don’t think that your clustering problem will work well with K-means clustering, check out these resources on alternative cluster modeling techniques:

* [**Sci-kit Clustering Modules**](http://scikit-learn.org/stable/modules/clustering.html)**–**this documentation has a nifty image that visually compares the clustering algorithms in scikit-learn, as they look for different scatterplots. Using this documentation can point you to the right algorithm to use if you have a scatter plot similar to one of their examples. It also gives you some insight on how to evaluate your clustering model mathematically.
* [**Clustering Algorithms**](http://web.stanford.edu/class/cs345a/slides/12-clustering.pdf) – this Powerpoint presentation from Stanford’s CS345 course, Data Mining, gives insight into different techniques – how they work, where they are effective and ineffective, etc. It is a great learning resource to understand how clustering works at a theoretical level.