Unit 4: Identifying Trends and Creating Models

Contents

Lab Questions

Getting Started

In previous units, we worked on loading, cleaning, and exploring data. While working with the data, we noted that certain relationships appeared to exist between columns/variables. While plots allowed us to make claims like "x increases as y decreases", we didn't try to model that relationship mathematically nor did we try to determine the quality of that model.

In this unit,we'll look at creating models for the relationships in our data; specifically, we'll look at linear models for numerical data and logistic models for categorical data.

To create and explore these models, we'll use the StatsModels (https://www.statsmodels.org/stable/index.html) library. To install it, we'll use `!pip` .

In [ ]:
```
!pip install statsmodels
```

We'll work with plots in this unit. To ensure they are displayed in the notebook itself, we'll need to use the `%matplotlib inline` command.

In [1]:
```
%matplotlib inline
```

In addition to the Seaborn and pandas libraries, we'll make explicit use of libraries on which they depend.

- matplotlib.pyplot (https://matplotlib.org/api/pyplot_api.html): a collection of plotting functions with MATLAB (https://www.mathworks.com/products/matlab.html)-like syntax
- numpy (http://www.numpy.org/): scientific computing library

We'll import these and StatsModels with names that follow standard convention.

We also set the figure size for plots and a marker size for outliers in box plots.

In [2]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
import statsmodels.api as sm

sns.set(rc={'figure.figsize':(12,8), "lines.markeredgewidth": 0.5 })
```

```
/usr/local/lib/python3.6/site-packages/statsmodels/compat/pandas.py:56: F
utureWarning: The pandas.core.datetools module is deprecated and will be
removed in a future version. Please use the pandas.tseries module instea
d.
  from pandas.core import datetools
```

Linear Models

The first type of model we'll look at are linear models also known as linear regressions. In basic terms, we use a linear model if the data looks like a line could be drawn through it. More specifically, for two dimensional data, we try to model the relationship between two variables, the independent or input variable, *X*, and the dependent or response variable, *Y*, by an equation of the form

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

where β_0 and β_1 are *coefficients*. We can generalize this to more than two dimensions. If $X_1, X_2, \ldots, X_n$ are independent variables, and Y is the dependent variable, we try to model the relationship by an equation in the form

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_n X_n$$

where $\beta_0, \beta_1, \ldots, \beta_n$ are the coefficients.

To find the coefficients of these equations, we'll rely on the ordinary least squares (https://en.wikipedia.org/wiki/Ordinary_least_squares) method that attempts to minimizing the the sum of squares of the differences between the observed values and the predicted values - we'll explore this further in a bit.

An Example

Let's look at an example of how we can calculate the coefficients of a linear equation that models some data. We'll start with an example based on the StatsModels documentation (http://www.statsmodels.org/stable/examples/notebooks/generated/ols.html).

First, we generate our "observed" data. To do this, we'll use the Numpy *linspace()* (https://docs.scipy.org/doc/numpy/reference/generated/numpy.linspace.html) function to generate 100 evenly-spaced values between 0 and 10; these will correspond to values that will be used for the independent variable. Next, we'll use the *normal()* (https://docs.scipy.org/doc/numpy/reference/generated/numpy.random.normal.html) function from NumPy's *random* (https://docs.scipy.org/doc/numpy-1.14.0/reference/routines.random.html) submodule to draw 100 samples from a normal distribution - this will simulate errors in our data.

In [3]:
```
nsample = 100
x = np.linspace(0, 10, nsample)
e = np.random.normal(size=nsample)

display(x[:10])
```

```
array([0.        , 0.1010101 , 0.2020202 , 0.3030303 , 0.4040404 ,
       0.50505051, 0.60606061, 0.70707071, 0.80808081, 0.90909091])
```

Looking at the first 10 values of `x`, we can see that they are stored in an array (https://docs.scipy.org/doc/numpy-1.14.0/reference/generated/numpy.array.html).

Next, we calculate our "observed" values as a combination of the independent variable, a constant, and some error.

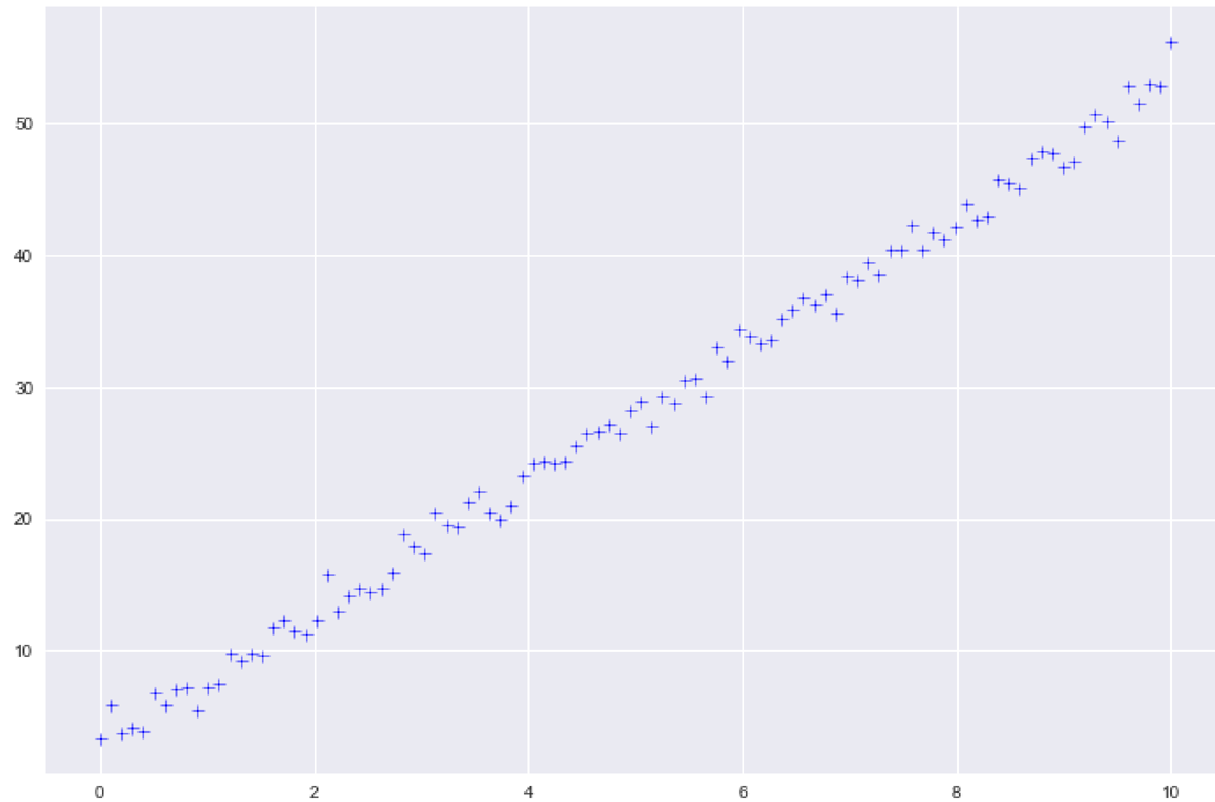
In [4]:
```
y = 5 * x + 3 + e
```

Let's plot the values of `x` and `y`. We'll create a scatter plot but use a different approach to do this.

First, we create figure (https://matplotlib.org/api/_as_gen/matplotlib.figure.Figure.html#matplotlib.figure.Figure) and axes (https://matplotlib.org/api/axes_api.html#matplotlib.axes.Axes) objects using the pyplot *subplots()* (https://matplotlib.org/api/_as_gen/matplotlib.pyplot.subplots.html) function; this is useful when we want to plot items from different sources together (as we will do in a bit). We use the axes' *plot()* (https://matplotlib.org/api/_as_gen/matplotlib.axes.Axes.plot.html#matplotlib.axes.Axes.plot) method to plot the coordinate pairs from `x` and `y`; the third argument indicates that we'd like to use blue plus-sign markers rather than draw lines from the data.

In [5]:
```
fig, ax = plt.subplots()
ax.plot(x, y, 'b+')
```

Out[5]:
```
[<matplotlib.lines.Line2D at 0x1096bfb70>]
```



In the two-dimensional linear regression. we need to calculate the values of two coefficients: a constant and a value that will be multiplied by the value of the independent variable. As we saw above, we can write the linear model in the form

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

We can write the constant, β_0 as $\beta_0 X^0$ Since any value raised to the zeroth power is one, we can write β_0 as $1 \cdot \beta_0$. We can say that the dependent variable is a linear combination of 1 and the independent variable. For our model, we account for this by using the StatsModels' [add_constant() (http://www.statsmodels.org/dev/generated/statsmodels.tools.tools.add_constant.html)](http://www.statsmodels.org/dev/generated/statsmodels.tools.tools.add_constant.html) function.

In [6]:
```
X = sm.add_constant(x)
display(X[:10])
```

```
array([[1.        , 0.        ],
       [1.        , 0.1010101 ],
       [1.        , 0.2020202 ],
       [1.        , 0.3030303 ],
       [1.        , 0.4040404 ],
       [1.        , 0.50505051],
       [1.        , 0.60606061],
       [1.        , 0.70707071],
       [1.        , 0.80808081],
       [1.        , 0.90909091]])
```

Comparing `X` to `x` we now have an array of arrays where the first element in the inner arrays are `1` corresponding to the value of the independent variable for the constant term.

To compute the the linear regression, we first set up the ordinary least squares model using the StatsModels' *OLS* (http://www.statsmodels.org/dev/generated/statsmodels.regression.linear_model.OLS.html) function and by specifying the array of values for the dependent variable and the array of values for the independent variable.

With the model created, we calculate the regression coefficients using the model's *fit()* (http://www.statsmodels.org/dev/generated/statsmodels.regression.linear_model.OLS.fit.html#statsm method; this method returns a *RegressionResults* (http://www.statsmodels.org/dev/generated/statsmodels.regression.linear_model.RegressionResults.h object. To dispaly information about the fit, we display the output of the result's *summary()* (http://www.statsmodels.org/dev/generated/statsmodels.regression.linear_model.RegressionResults.s method.

In [7]:
```
model = sm.OLS(y, X)
results = model.fit()
display(results.summary())
```

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.995
Model:	OLS	**Adj. R-squared:**	0.995
Method:	Least Squares	**F-statistic:**	2.016e+04
Date:	Wed, 04 Apr 2018	**Prob (F-statistic):**	2.86e-115
Time:	22:42:26	**Log-Likelihood:**	-144.10
No. Observations:	100	**AIC:**	292.2
Df Residuals:	98	**BIC:**	297.4
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	2.8845	0.205	14.070	0.000	2.478	3.291
x1	5.0285	0.035	141.974	0.000	4.958	5.099

Omnibus:	1.344	**Durbin-Watson:**	1.949
Prob(Omnibus):	0.511	**Jarque-Bera (JB):**	1.145
Skew:	0.262	**Prob(JB):**	0.564
Kurtosis:	2.984	**Cond. No.**	11.7

From the results, the coefficients, their p-values, and the value of `R-squared` are particularly of interest. We can access these directly from `results` using the *params*, *pvalues*, and *rsquared* properties as well.

In [8]:
```
print('Parameters: ', results.params)
print("P-values:", results.pvalues)
print('R^2: ', results.rsquared)
```

```
Parameters:  [2.88448741 5.02848078]
P-values: [2.94333430e-025 2.86173287e-115]
R^2:  0.9951616020106396
```

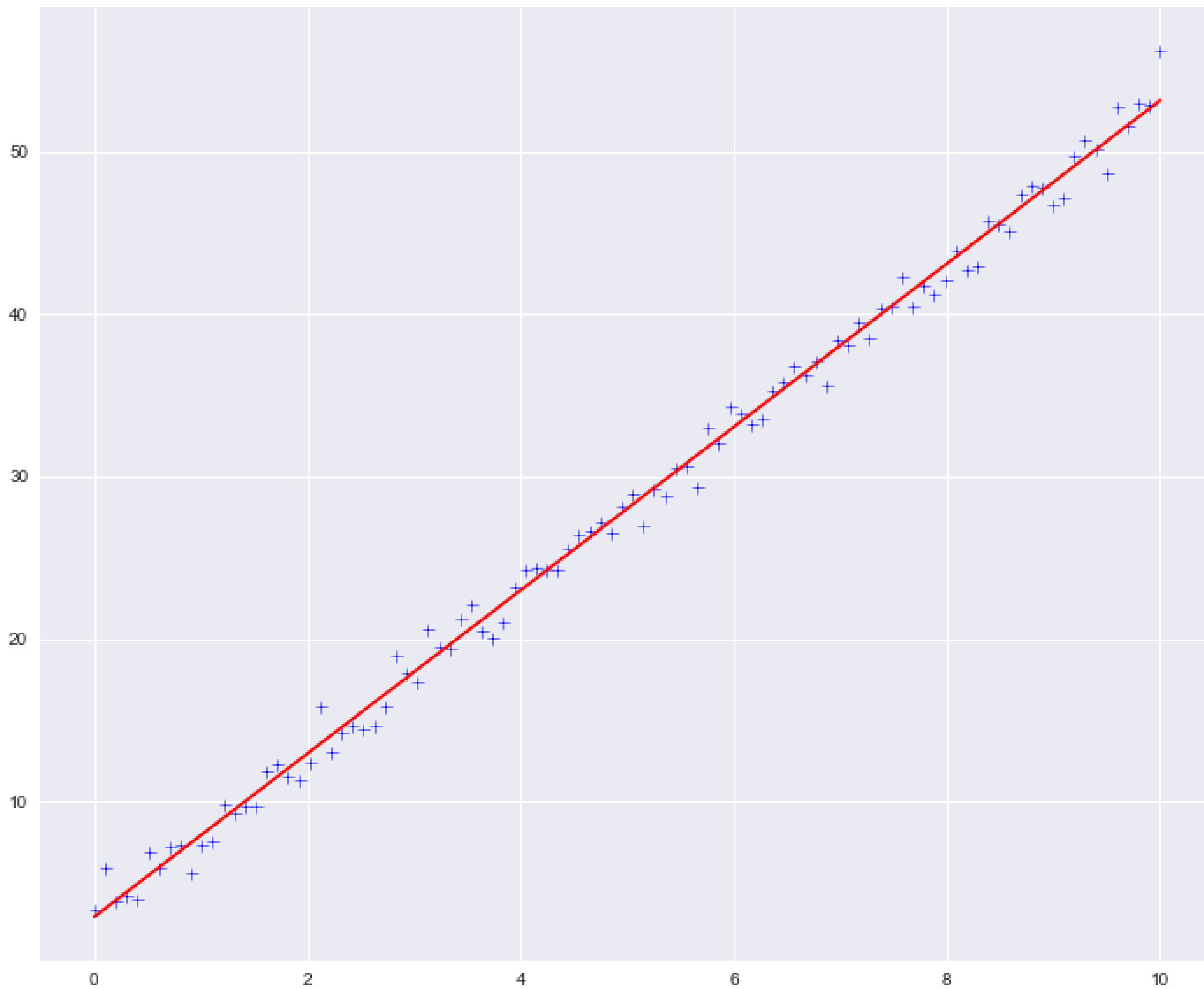
The coefficients appear in the same order in which the independent variables appear in `X` .

Note that this is "close" to the equation we used to generate the data - the discrepancy is due to the error we introduced.

Let's plot the regression line along with our data. We can repeat the same steps as before to create the scatter plot. We make an additional call to *plot()* and specify the y-values as the output from the results *predict()* method which returns the predicted values of the dependent variable based on the regression results. Specifying `'r'` for the third argument to plot indicates that we would like the line to be red.

In [9]:
```
fig, ax = plt.subplots(figsize=(12,10))
ax.plot(x, y, 'b+')
ax.plot(x, results.predict(), 'r')
```

Out[9]: [<matplotlib.lines.Line2D at 0x10dce0eb8>]



The p-value (https://en.wikipedia.org/wiki/P-value) associated with each coefficient indicates how likely changes to the corresponding independent variable account for changes in the dependent variable. A p-value close to zero (typically, less than 0.05) indicates that the independent variable provides a meaningful addition to the model. [1]

The `R-squared` value is also known as the coefficient of determination (https://en.wikipedia.org/wiki/Coefficient_of_determination) and provides a measure of how well the regression line fits the data. The coefficient of determination can range from 0 to 1 with 0 indicating (in some regressions, the value can be negative) and indicates how much of the variation in dependent variable can be explained by the model - a value of 1 indicates that all variation is explained by the model and 0 indicates that the model accounts for none of the variation. In this example, some of the variation is due to the error we introduced - changing the magnitude of the error or the parameters of the distribution from which values were drawn will result in a better or worse coefficient of determination.

Simple Linear Regression

Let's look at an example based on real data. To begin, we'll load the fuel economy data we processed previously.

In [10]:
```
epa_data = pd.read_csv("./data/02-vehicles.csv", engine="python")
```

Recall that a description of the data is available in `./data/02-vehicles-description.html`.

In [11]:
```
from IPython.display import HTML
HTML(filename="./data/02-vehicles-description.html")
```

- eng_dscr - engine descriptor; see http://www.fueleconomy.gov/feg/findacarhelp.shtml#engine
- evMotor - electric motor (kw-hrs)
- feScore - EPA Fuel Economy Score (-1 = Not available)
- fuelCost08 - annual fuel cost for fuelType1 ($)
- fuelCostA08 - annual fuel cost for fuelType2 ($)
- fuelType - fuel type with fuelType1 and fuelType2 (if applicable)
- fuelType1 - fuel type 1. For single fuel vehicles, this will be the only fuel. For dual fuel vehicles, this will be the conventional fuel.
- fuelType2 - fuel type 2. For dual fuel vehicles, this will be the alternative fuel (e.g. E85, Electricity, CNG, LPG). For single fuel vehicles, this field is not used
- ghgScore - EPA GHG score (-1 = Not available)
- ghgScoreA - EPA GHG score for dual fuel vehicle running on the alternative fuel (-1 = Not available)
- guzzler- if G or T, this vehicle is subject to the gas guzzler tax
- highway08 - highway MPG for fuelType1
- highway08U - unrounded highway MPG for fuelType1
- highwayA08 - highway MPG for fuelType2

For this unit, we'll work the the following columns.

- `co2`
- `comb08`
- `cylinders`
- `displ`
- `highway08`
- `city08`

We can also remove rows with missing or non-positive values. Because the DataFrame consists of only numeric data, we can use a sort of shortcut for removing rows with non-positive values. We create a mask applied to the entire DataFrame rather than a specific column - this applies it to all columns. We then use *all()* with an argument of 1 to indicate that the property applies across all columns. Effectively this mask will match any row in which all the values are positive.

In [12]:
```
epa_subset = epa_data[['co2', 'comb08', 'cylinders', 'displ', 'highway08',
epa_subset = epa_subset[(epa_subset > 0).all(1)].copy()
epa_subset.head()
```

Out[12]:

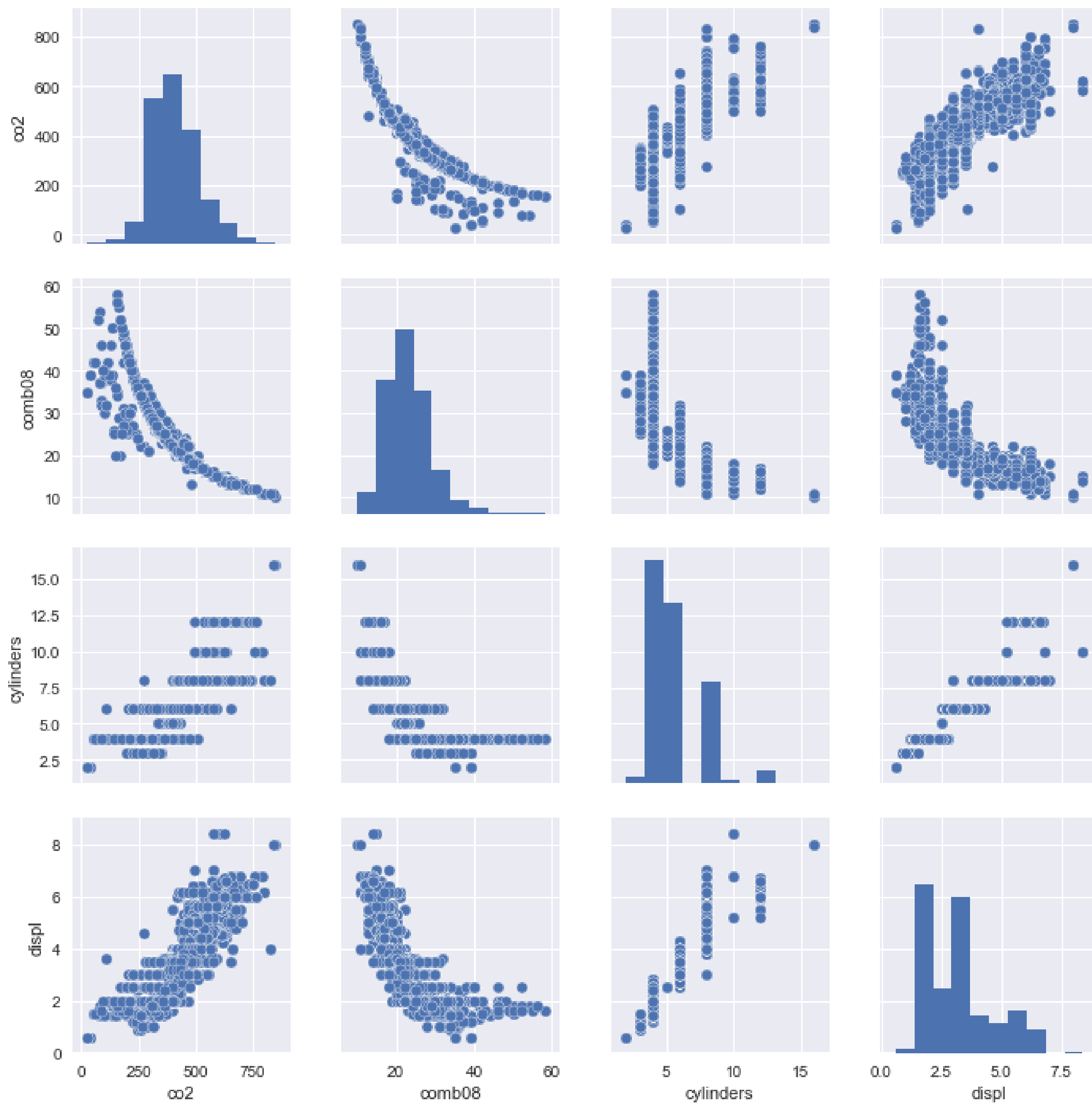
	co2	comb08	cylinders	displ	highway08	city08
16780	318	32	4.0	2.0	37	29
16781	315	32	4.0	2.0	39	29
16839	318	32	4.0	2.0	37	29
16840	315	32	4.0	2.0	39	29
21337	315	32	4.0	2.0	39	29

To start, let's look at a pair plot for the `co2` , `comb08` , `cylinders` , and `displ` columns in `epa_subset` .

Lab 1 In the cell below, create a pair plot for the `co2` , `comb08` , `cylinders` , and `displ` columns in `epa_subset` .

```
In [13]: sns.pairplot(data=epa_subset[['co2', 'comb08', 'cylinders', 'displ']])

Out[13]: <seaborn.axisgrid.PairGrid at 0x107de16a0>
```

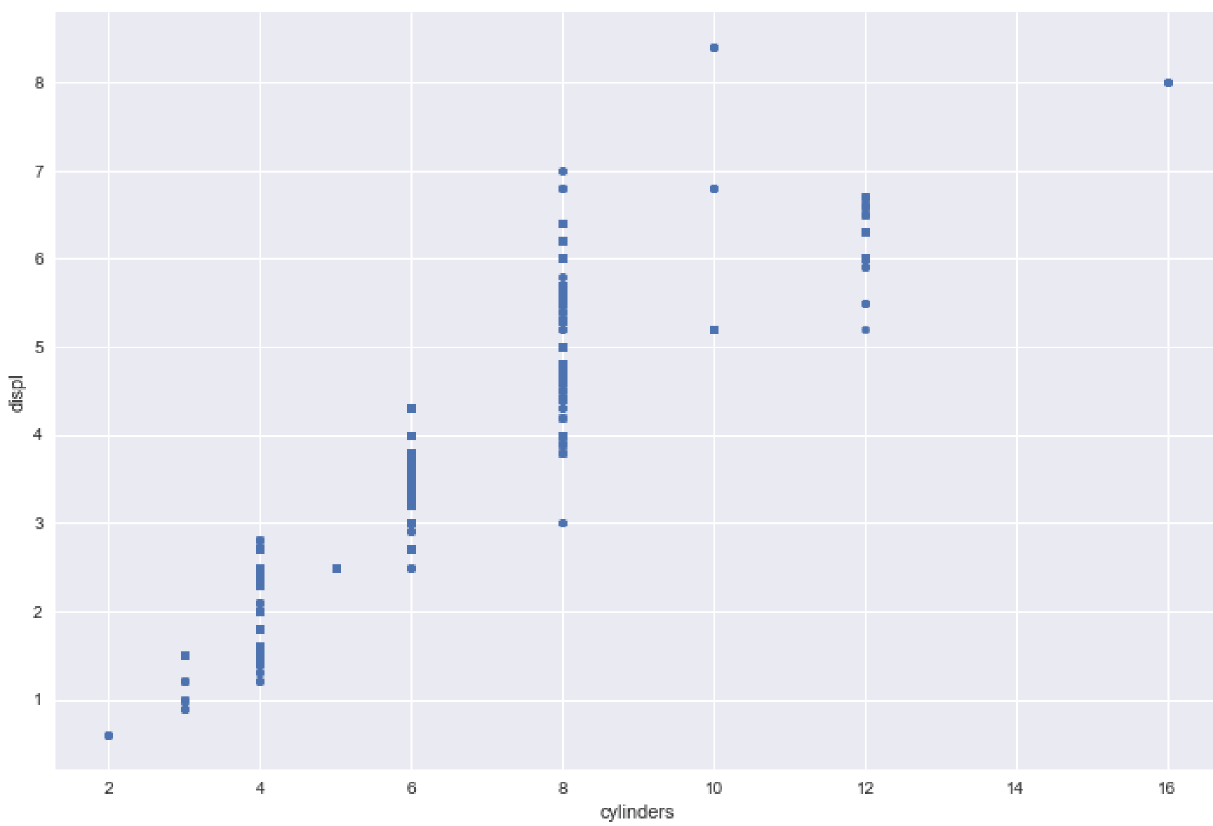


Let's look more closely at the relationship between `Cylinders` and `Displacement`.

Lab 2 In the cell below, create scatter plot for the `cylinders` and `displacement` columns in `epa_subset`.

In [14]:
```
epa_subset.plot.scatter(x='cylinders', y='displ')
```

Out[14]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x10f9c3f28>
```



Next, lets create the least squares model and fit a line to the data.

In [16]:
```
X = sm.add_constant(epa_subset.cylinders)
Y = epa_subset.displ
model = sm.OLS(Y, X)
res = model.fit()
display(res.summary())
```

OLS Regression Results

Dep. Variable:	displ	**R-squared:**	0.859
Model:	OLS	**Adj. R-squared:**	0.859
Method:	Least Squares	**F-statistic:**	4.527e+04
Date:	Wed, 04 Apr 2018	**Prob (F-statistic):**	0.00
Time:	22:42:46	**Log-Likelihood:**	-5593.4
No. Observations:	7402	**AIC:**	1.119e+04
Df Residuals:	7400	**BIC:**	1.120e+04
Df Model:	1		
Covariance Type:	nonrobust		

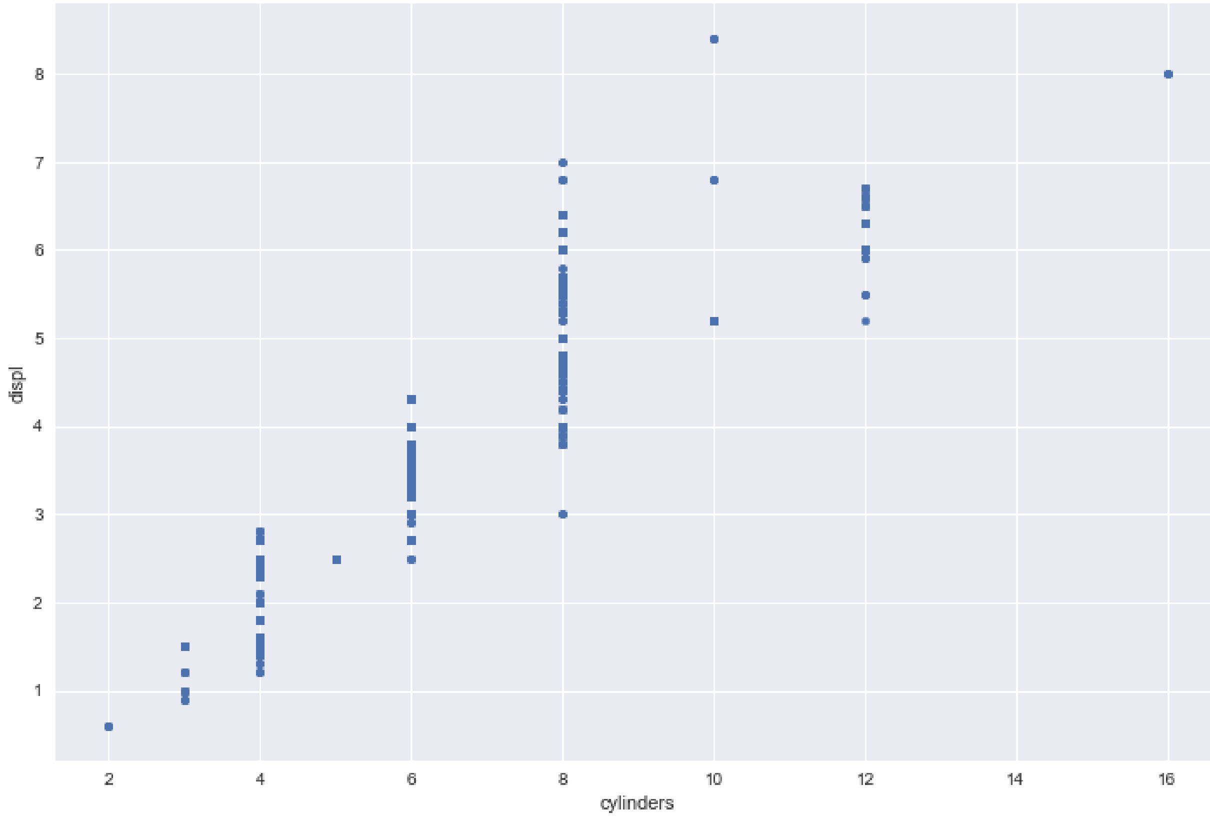
	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	-0.6751	0.019	-35.240	0.000	-0.713	-0.638
cylinders	0.6834	0.003	212.759	0.000	0.677	0.690

Omnibus:	488.472	**Durbin-Watson:**	0.995
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1327.138
Skew:	0.364	**Prob(JB):**	6.54e-289
Kurtosis:	4.942	**Cond. No.**	19.6

From the `R-squared` value we can see there is a somewhat strong linear relationship between the data; further, the p-value for the coefficient of `cylinders` indicates that changes in `dislp` are likely attributed to changes in `cylinders` .

To plot the fit line with the scatter plot generated by the DataFrame's *plot()* method we can use StatsModel's *abline_plot()* (http://www.statsmodels.org/dev/generated/statsmodels.graphics.regressionplots.abline_plot.html) function. First, we import the function then create a scatter plot and store the returned *axes* object.

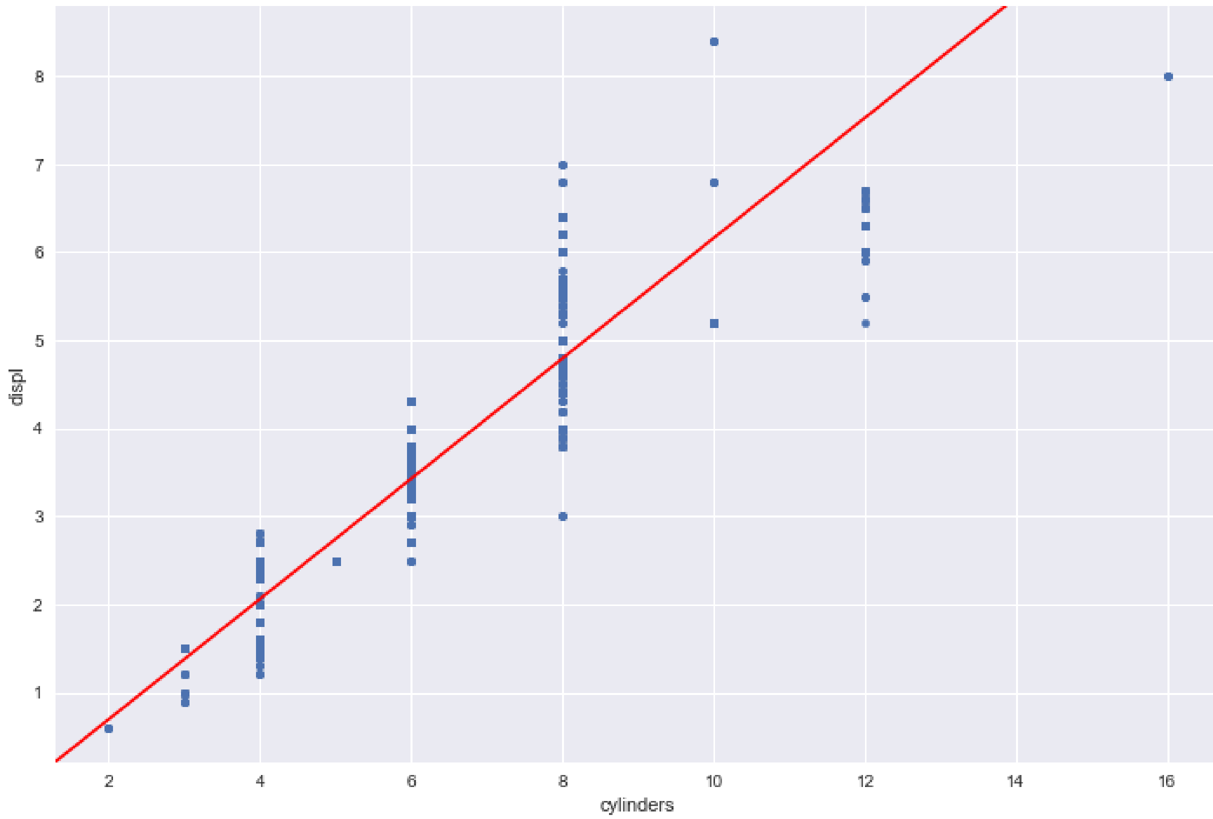
In [17]:
```
from statsmodels.graphics.regressionplots import abline_plot
axes = epa_subset.plot.scatter(x="cylinders", y="displ")
```



We can can add the plot of the regression line to the plot using *abline_plot()* function, specifying the model results, the *axes*, and a color.

In [18]:
```
abline_plot(model_results=res, ax=axes, color='r')
```

Out[18]:

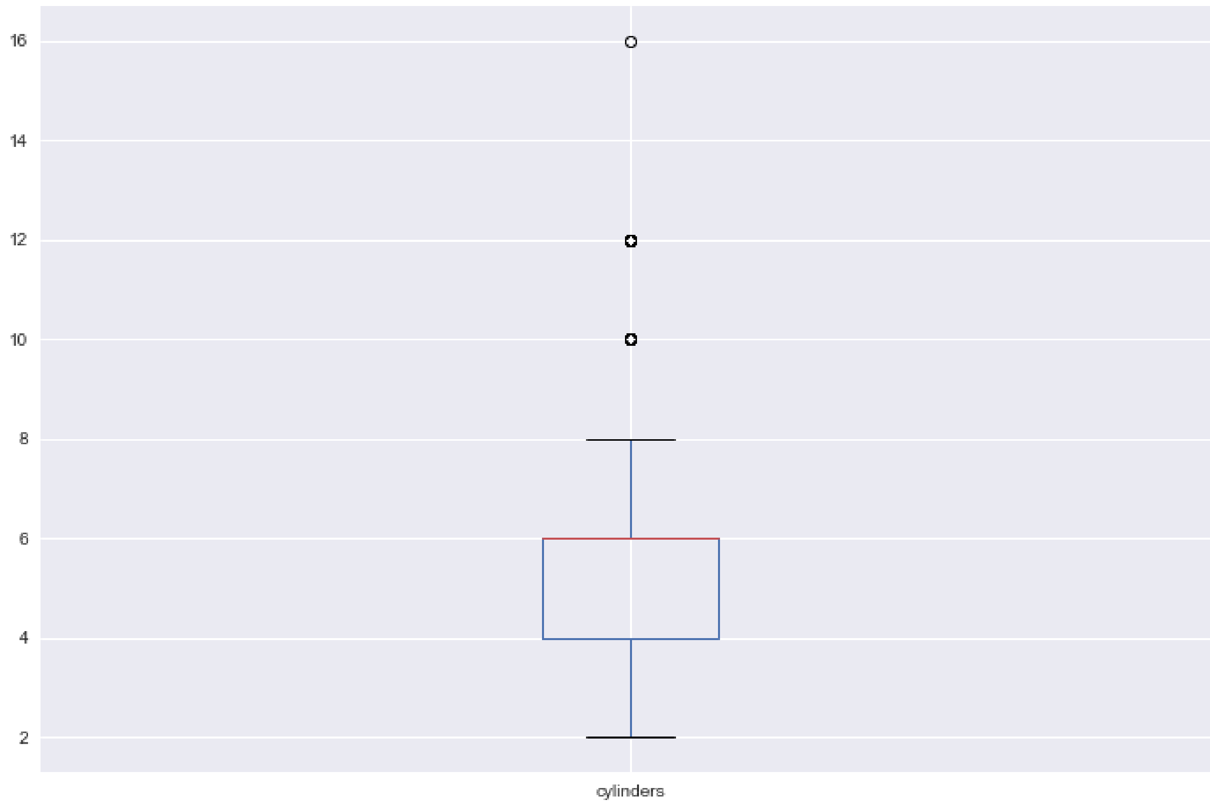


Often when creating models, we want to exclude outliers in our calculations. Let's look at the box plots for `cylinders` and `displ` .

Lab 3 In the cells below, create the box plots for `cylinders` and `displ` .

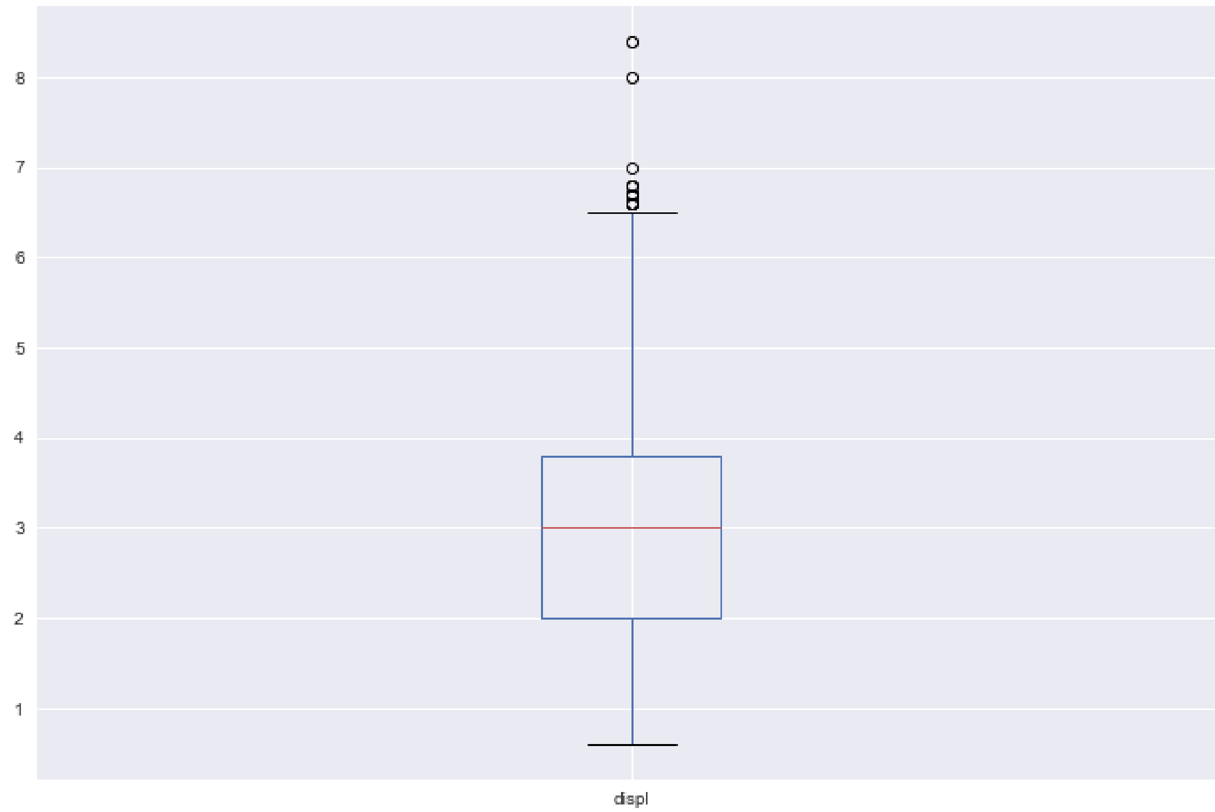
In [19]:
```
epa_subset.cylinders.plot(kind='box')
```

Out[19]: <matplotlib.axes._subplots.AxesSubplot at 0x1100dccf8>



In [20]:
```
epa_subset.displ.plot(kind='box')
```

Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x1102623c8>



We can see that there are a few outliers for each column. We can remove them and recalculate the fit. We can create a copy of the DataFrame from which will will remove outliers.

In [21]:
```
epa_no_outliers = epa_subset[['cylinders', 'displ']].copy()
```

To remove the outliers from a column, we'll create a function that we can apply to a DataFrame. We'll define outliers based on the interquartile range.

After defining the function, we can use it with our `epa_no_outliers` DataFrame.

In [22]:
```
def remove_outliers(dataframe, column):
    q1 = dataframe[column].quantile(0.25)
    q3 = dataframe[column].quantile(0.75)
    iqr = q3 - q1
    lower = q1 - 1.5 * iqr
    upper = q3 + 1.5 * iqr
    return dataframe[(dataframe[column] >= lower) &
                     (dataframe[column] <= upper)].copy()

epa_no_outliers = remove_outliers(epa_no_outliers, "cylinders")
epa_no_outliers = remove_outliers(epa_no_outliers, "displ")
```

With the outliers removed, we can create a new model and calculate the regression coefficients.

Lab 4 In the cell below, create an ordinary least squares model where `cylinders` is the independent variable and `displ` is the dependent variable. Include a coefficient term in the model. Calculate the fit and store the result in a variable named `res_no_outliers`.

In [24]:
```
X = sm.add_constant(epa_no_outliers.cylinders)
Y = epa_no_outliers.displ
model = sm.OLS(Y, X)
res_no_outliers = model.fit()
```

Looking at the result's summary and the `R-squared` value, we can see the the fit is slightly better with the outliers removed.

In [25]:
```
display(res_no_outliers.summary())
```

OLS Regression Results

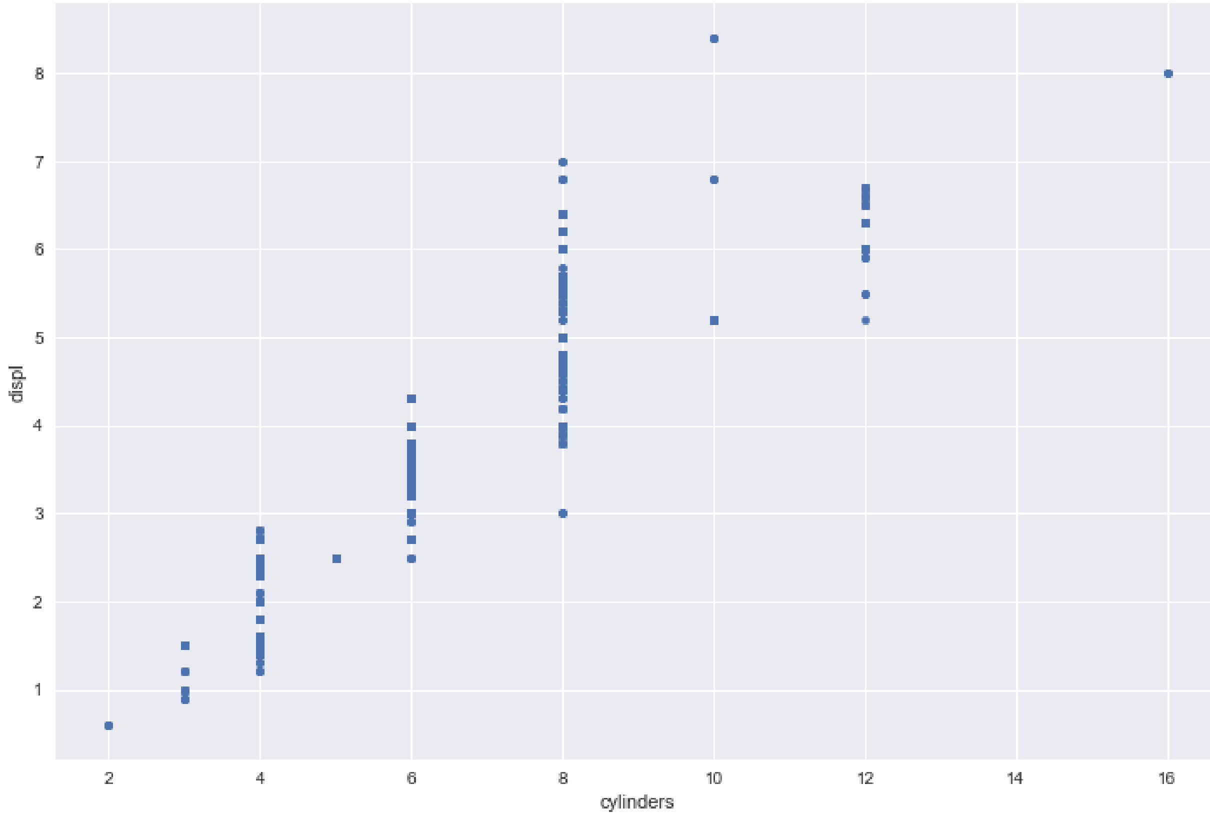
Dep. Variable:	displ	**R-squared:**	0.883
Model:	OLS	**Adj. R-squared:**	0.883
Method:	Least Squares	**F-statistic:**	5.384e+04
Date:	Wed, 04 Apr 2018	**Prob (F-statistic):**	0.00
Time:	22:43:06	**Log-Likelihood:**	-4103.9
No. Observations:	7132	**AIC:**	8212.
Df Residuals:	7130	**BIC:**	8225.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	-1.1141	0.019	-59.314	0.000	-1.151	-1.077
cylinders	0.7681	0.003	232.042	0.000	0.762	0.775

Omnibus:	104.759	**Durbin-Watson:**	1.014
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	153.838
Skew:	0.167	**Prob(JB):**	3.93e-34
Kurtosis:	3.637	**Cond. No.**	21.5

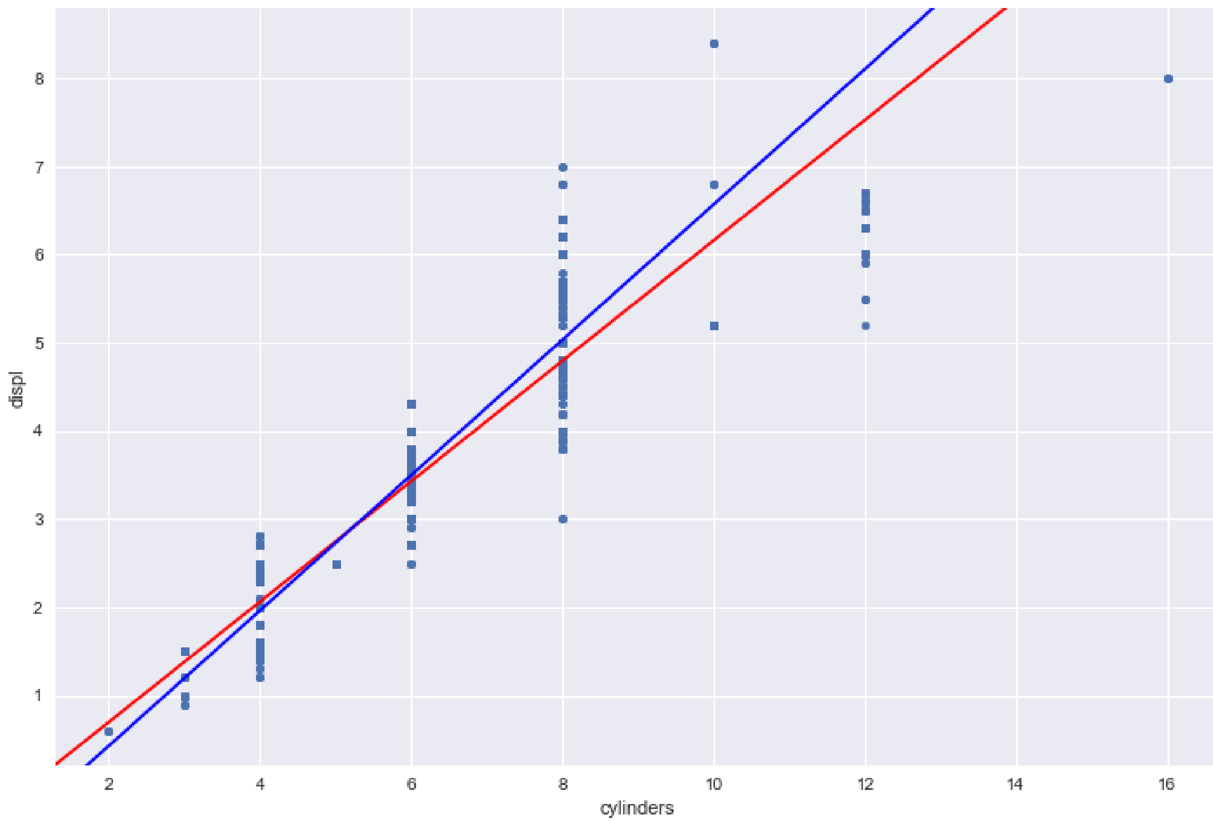
We can plot both the original fit and the fit calculated without outliers against a scatter plot of the data.

In [26]:
```
axes = epa_subset.plot.scatter(x="cylinders", y="displ")
```



In [27]:
```
abline_plot(model_results=res, ax=axes, color='r')
abline_plot(model_results=res_no_outliers, ax=axes, color='b')
```

Out[27]:



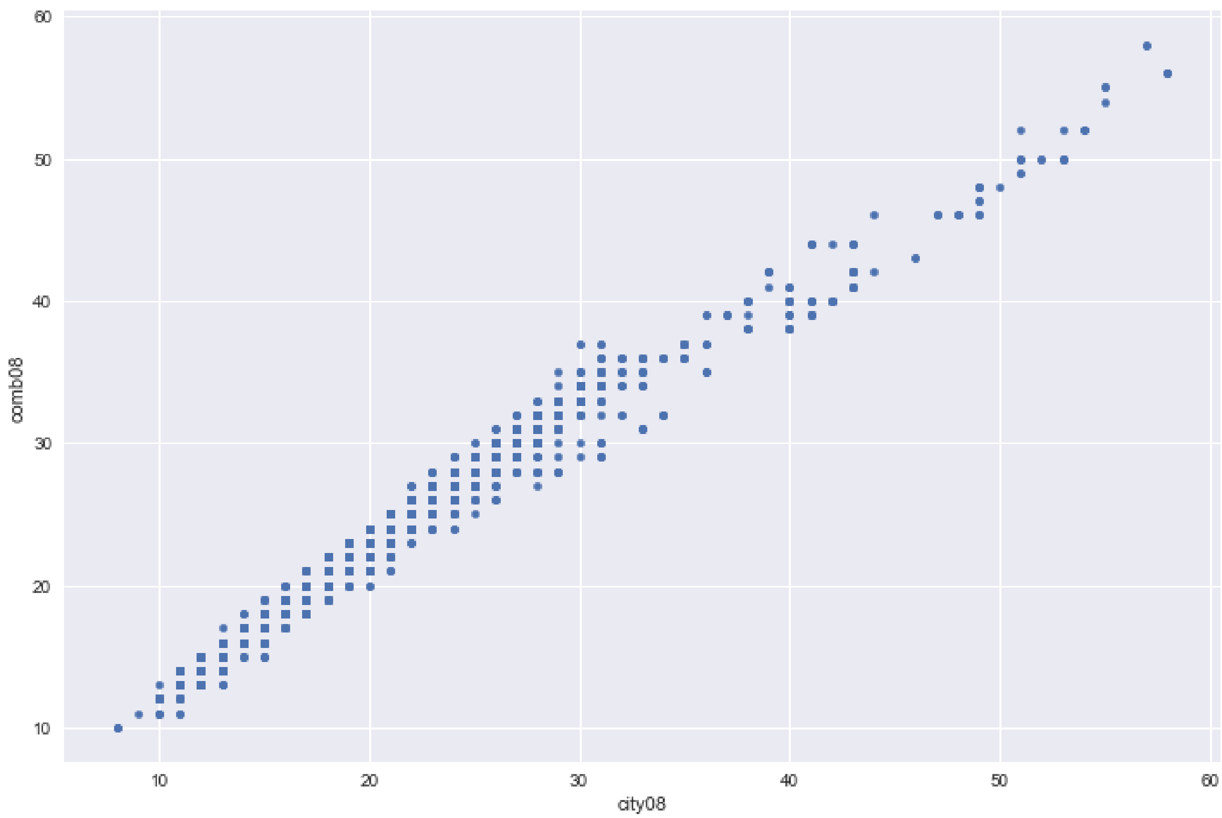
The original regression line is colored red and the regression line calculated with outliers removed is colored blue.

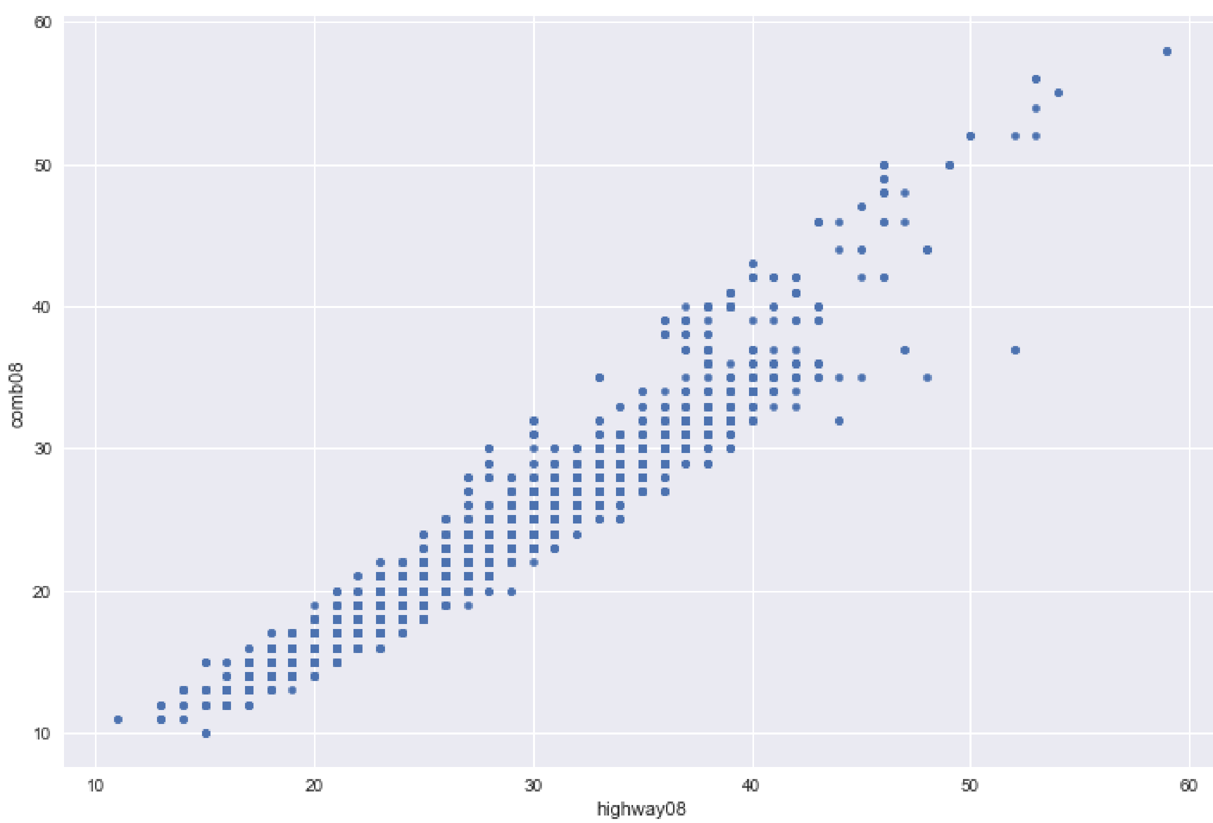
Multiple Linear Regression

So far, we've looked at regressions in which there is one independent variable and a constant. Often, changes in the response variable are dependent on multiple variables. For example. we expect that the combined fuel economy is dependent on both city and highway economy. We can see from the scatter plots that `comb08` looks linearly dependent on `city08` and `highway08`.

In [28]:
```
epa_subset.plot.scatter(x='city08', y='comb08')
epa_subset.plot.scatter(x='highway08', y='comb08')
```

Out[28]: <matplotlib.axes._subplots.AxesSubplot at 0x1103a6ba8>





We can also fit models for each of these individually - `city08` / `comb08` and highway08 / comb08 .

In [29]:
```
X = sm.add_constant(epa_subset.city08)
Y = epa_subset.comb08
model = sm.OLS(Y, X)
res = model.fit()
display(res.summary())
```

OLS Regression Results

Dep. Variable:	comb08	**R-squared:**	0.972
Model:	OLS	**Adj. R-squared:**	0.972
Method:	Least Squares	**F-statistic:**	2.590e+05
Date:	Wed, 04 Apr 2018	**Prob (F-statistic):**	0.00
Time:	22:43:14	**Log-Likelihood:**	-10242.
No. Observations:	7402	**AIC:**	2.049e+04
Df Residuals:	7400	**BIC:**	2.050e+04
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
const	2.9449	0.041	71.791	0.000	2.865	3.025
city08	0.9854	0.002	508.942	0.000	0.982	0.989

Omnibus:	1685.615	**Durbin-Watson:**	1.292
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	6373.261
Skew:	-1.096	**Prob(JB):**	0.00
Kurtosis:	6.982	**Cond. No.**	77.6

In [30]:
```
X = sm.add_constant(epa_subset.highway08)
Y = epa_subset.comb08
model = sm.OLS(Y, X)
res = est.fit()
display(res.summary())
```

OLS Regression Results

Dep. Variable:	displ	**R-squared:**	0.883
Model:	OLS	**Adj. R-squared:**	0.883
Method:	Least Squares	**F-statistic:**	5.384e+04
Date:	Wed, 04 Apr 2018	**Prob (F-statistic):**	0.00
Time:	22:43:15	**Log-Likelihood:**	-4103.9
No. Observations:	7132	**AIC:**	8212.
Df Residuals:	7130	**BIC:**	8225.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	-1.1141	0.019	-59.314	0.000	-1.151	-1.077
cylinders	0.7681	0.003	232.042	0.000	0.762	0.775

Omnibus:	104.759	**Durbin-Watson:**	1.014
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	153.838
Skew:	0.167	**Prob(JB):**	3.93e-34
Kurtosis:	3.637	**Cond. No.**	21.5

In [31]:
```
X = sm.add_constant(epa_subset[['city08', 'displ', 'comb08']])
Y = epa_subset.co2
model = sm.OLS(Y, X)
res = model.fit()
display(res.summary())
```

OLS Regression Results

Dep. Variable:	co2	**R-squared:**	0.907
Model:	OLS	**Adj. R-squared:**	0.907
Method:	Least Squares	**F-statistic:**	2.400e+04
Date:	Wed, 04 Apr 2018	**Prob (F-statistic):**	0.00
Time:	22:43:16	**Log-Likelihood:**	-35989.
No. Observations:	7402	**AIC:**	7.199e+04
Df Residuals:	7398	**BIC:**	7.201e+04
Df Model:	3		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	632.8284	3.828	165.301	0.000	625.324	640.333
city08	5.9879	0.386	15.531	0.000	5.232	6.744
displ	22.5756	0.432	52.236	0.000	21.728	23.423
comb08	-18.1745	0.410	-44.337	0.000	-18.978	-17.371

Omnibus:	1297.654	**Durbin-Watson:**	1.210
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	29733.488
Skew:	0.110	**Prob(JB):**	0.00
Kurtosis:	12.816	**Cond. No.**	339.

Separately, the models fit the data quite well. Let's look at the model in which both `city08` and `highway08` are independent variables.

In [32]:
```
X = sm.add_constant(epa_subset[['city08', 'highway08']])
Y = epa_subset.comb08
model = sm.OLS(Y, X)
res = model.fit()
display(res.summary())
```

OLS Regression Results

Dep. Variable:	comb08	**R-squared:**	0.996
Model:	OLS	**Adj. R-squared:**	0.996
Method:	Least Squares	**F-statistic:**	8.489e+05
Date:	Wed, 04 Apr 2018	**Prob (F-statistic):**	0.00
Time:	22:43:18	**Log-Likelihood:**	-3370.7
No. Observations:	7402	**AIC:**	6747.
Df Residuals:	7399	**BIC:**	6768.
Df Model:	2		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	-0.0860	0.022	-3.874	0.000	-0.130	-0.042
city08	0.6466	0.002	347.731	0.000	0.643	0.650
highway08	0.3600	0.002	199.909	0.000	0.356	0.364

Omnibus:	101.613	**Durbin-Watson:**	1.802
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	62.826
Skew:	-0.059	**Prob(JB):**	2.28e-14
Kurtosis:	2.564	**Cond. No.**	177.

Looking at the coefficient of determination, we can see this model, which depends on both `city08` and `highway08` , fits the data better than a model which depends on only one of the variables.

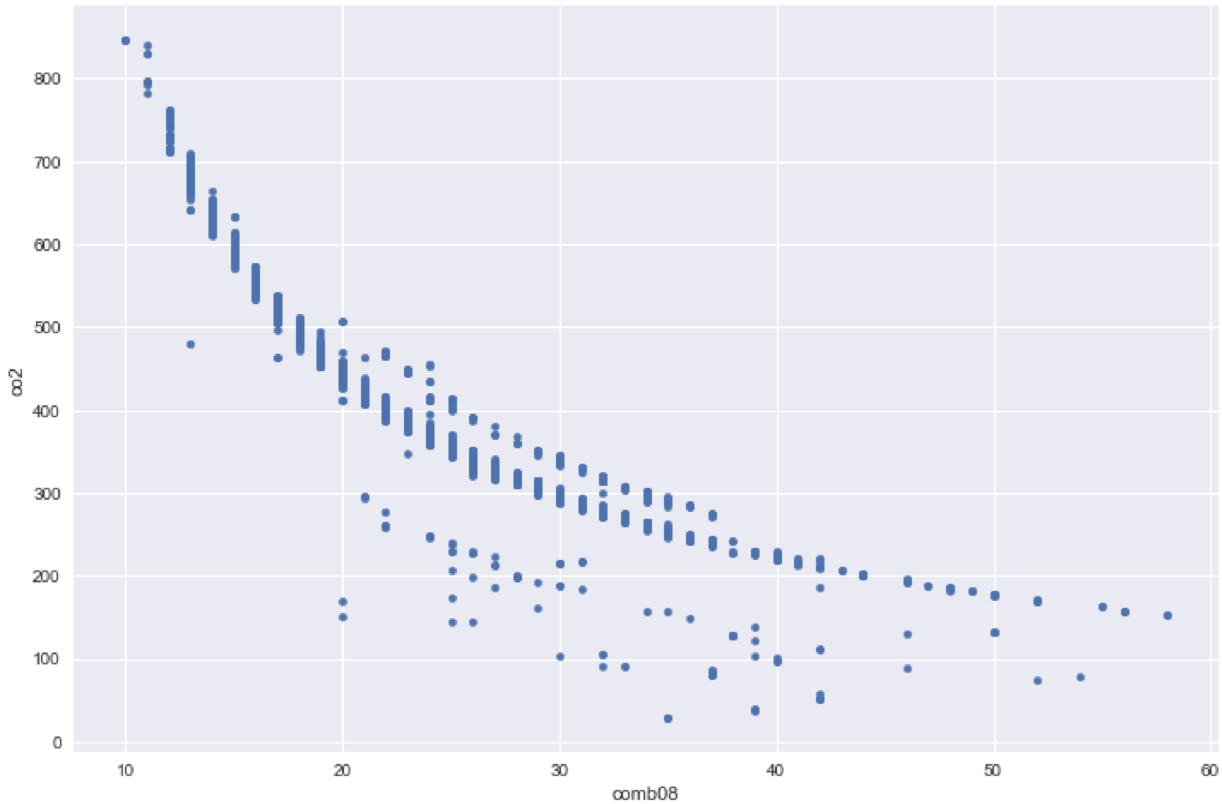
Linear-Like Regression

While there are many of relationships that are linear and that can be modeled using a linear regression, there are also relationships that are non-linear. Among these non-linear relationships are those that can transformed into a linear ones in terms of the coefficients and independent variables.

As an example, consider the relationship between `comb08` and `c02` .

In [33]:
```
epa_subset.plot.scatter(x="comb08", y="co2")
```

Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x11059f518>



While this relationship doesn't appear to be linear, it does look [hyperbolic (https://en.wikipedia.org/wiki/Hyperbola)](https://en.wikipedia.org/wiki/Hyperbola). In this case, the relationship between the independent variable and dependent variable could be written as

$$Y = \frac{1}{\beta_0 + \beta_1 X}$$

To move the coefficients and independent variable out of the denominator, we can take the reciprocal of both sides (provided the neither side is zero) - this gives us

$$\frac{1}{Y} = \beta_0 + \beta_1 X$$

For a given observation, this form is easier to work with since $\frac{1}{Y}$ and X are constants and we need to solve for β_0 and β_1.

We can calculate the reciprocal of the dependent variable, `co2` , and store the value in a new column.

In [34]:
```
epa_non_linear = epa_subset[['comb08', 'co2']].copy()
```

To transform the problem into a linear one, we need to calculate the reciprocal of the `co2` values.

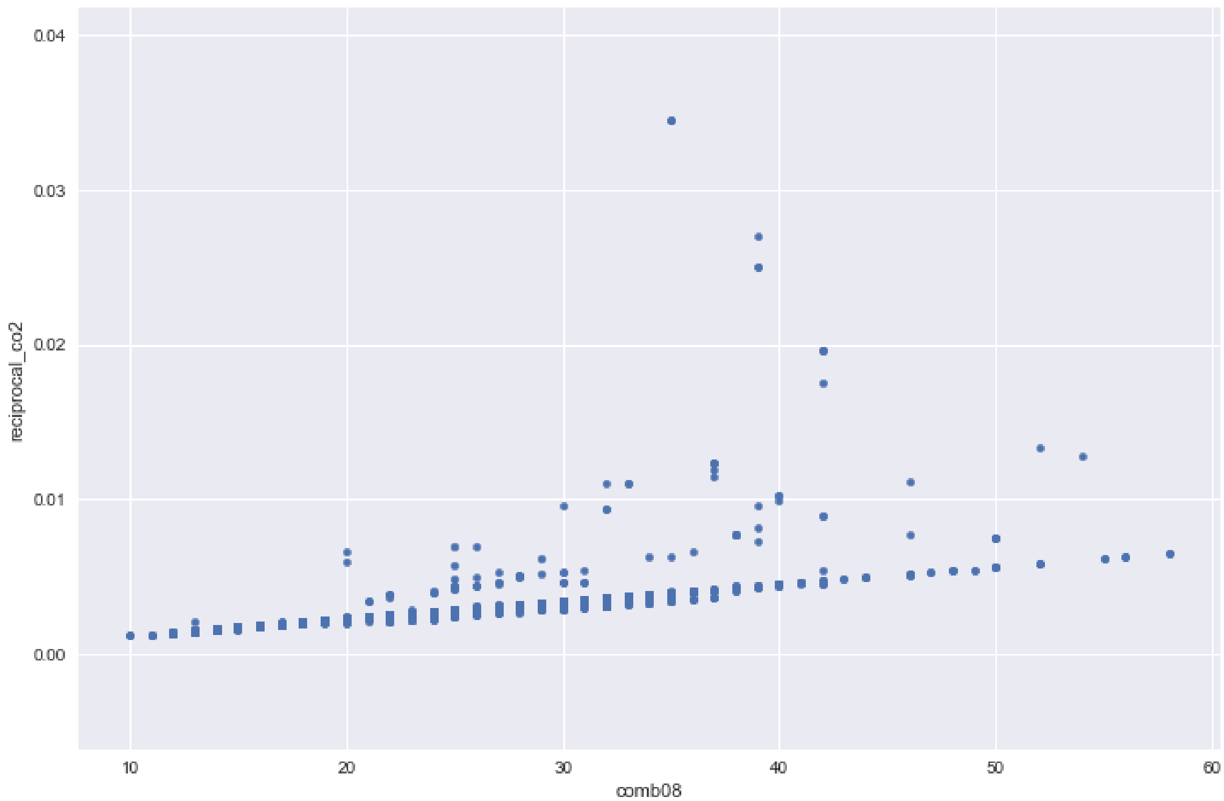
In [35]:
```
epa_non_linear["reciprocal_co2"] = 1/epa_non_linear.co2
```

Looking at the plot of `comb08` and `reciprocal_c02` , it appears that the relationship is linear, which supports our assumption that the original relationship was hyperbolic.

Lab 5 In the cell below, create a scatter plot of `comb08` and `reciprocal_c02` .

In [36]:
```
epa_non_linear.plot.scatter(x="comb08", y="reciprocal_co2")
```

Out[36]: `<matplotlib.axes._subplots.AxesSubplot at 0x110311f60>`



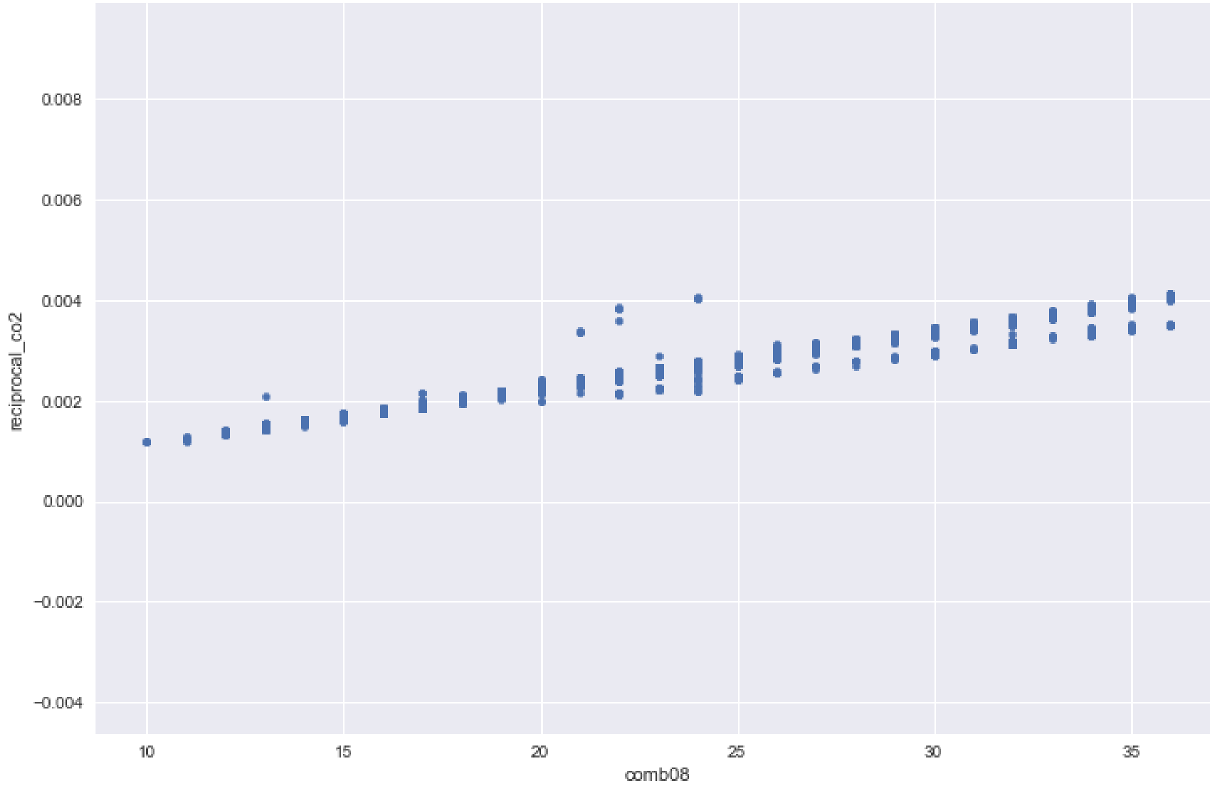
Before fitting the data with a linear model, let's remove the outliers.

Lab 6 In the cell below, remove the outliers from the `comb08` and `reciprocal_co2` columns in the `epa_non_linear` DataFrame. Use the *remove_outliers()* function we created earlier.

In [37]:
```
epa_non_linear = remove_outliers(epa_non_linear, "comb08")
epa_non_linear = remove_outliers(epa_non_linear, "reciprocal_co2")
```

With the outliers removed, let's look at the scatter plot again.

In [38]:
```
axes = epa_non_linear.plot.scatter(x="comb08", y="reciprocal_co2")
```



We can now create a linear model for `comb08` and `reciprocal_c02`. After calculating the coefficients, we display the summary of the results.

In [39]:
```
X = sm.add_constant(epa_non_linear["comb08"])
Y = epa_non_linear["reciprocal_co2"]
model = sm.OLS(Y, X)
res = model.fit()
display(res.summary())
```

OLS Regression Results

Dep. Variable:	reciprocal_co2	**R-squared:**	0.975
Model:	OLS	**Adj. R-squared:**	0.975
Method:	Least Squares	**F-statistic:**	2.804e+05
Date:	Wed, 04 Apr 2018	**Prob (F-statistic):**	0.00
Time:	22:43:38	**Log-Likelihood:**	56992.
No. Observations:	7193	**AIC:**	-1.140e+05
Df Residuals:	7191	**BIC:**	-1.140e+05
Df Model:	1		
Covariance Type:	nonrobust		

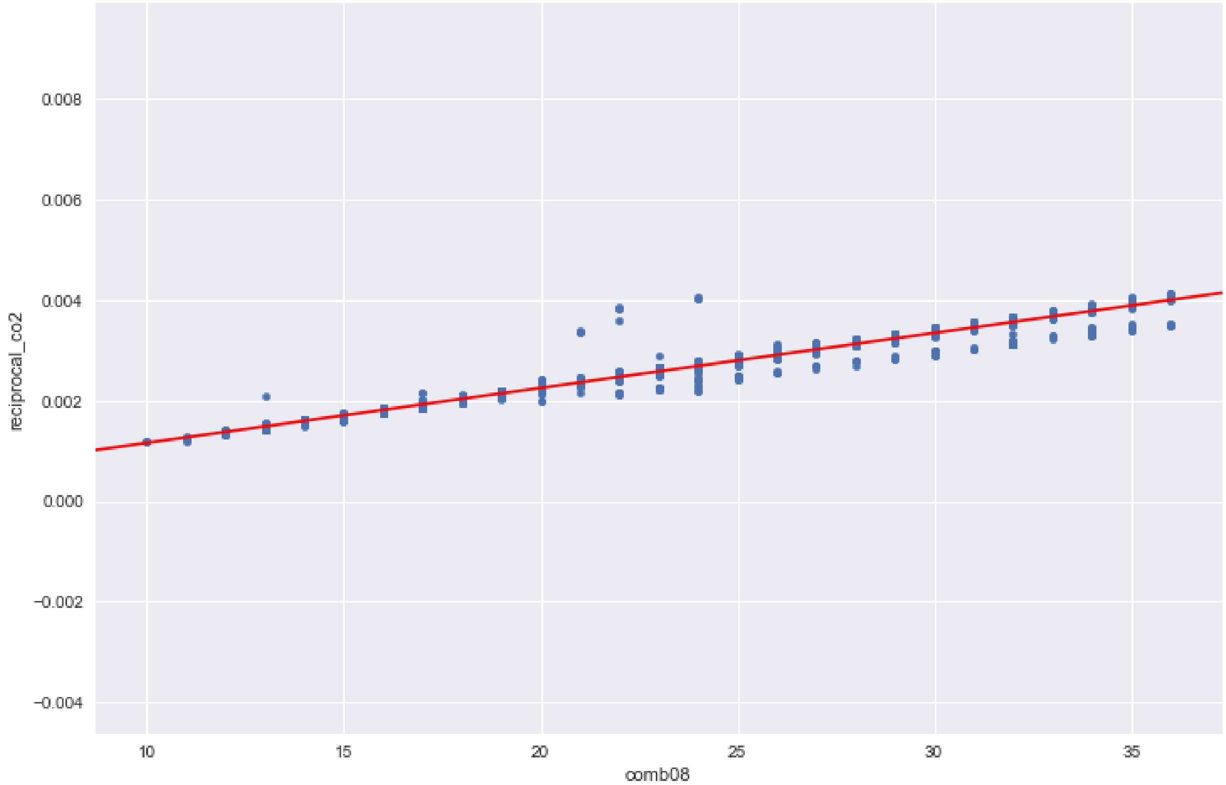
	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
const	6.764e-05	4.77e-06	14.182	0.000	5.83e-05	7.7e-05
comb08	0.0001	2.06e-07	529.486	0.000	0.000	0.000

Omnibus:	4638.856	**Durbin-Watson:**	1.511
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	1442146.541
Skew:	1.921	**Prob(JB):**	0.00
Kurtosis:	72.261	**Cond. No.**	107.

From the coefficient of determination, we see that the model produced a good fit. Adding the regression line to the existing scatter plot give the following plot.

In [40]:
```
abline_plot(model_results=res, ax=axes, color='r')
```

Out[40]:



We now need to transform the regression line to fit the original data. Using the *params* property of the results we have the following constant term and coefficient.

In [41]:
```
res.params
```

Out[41]:
```
const     0.000068
comb08    0.000109
dtype: float64
```

This means that our model is

$$Y = \frac{1}{0.000068+0.000109X}$$

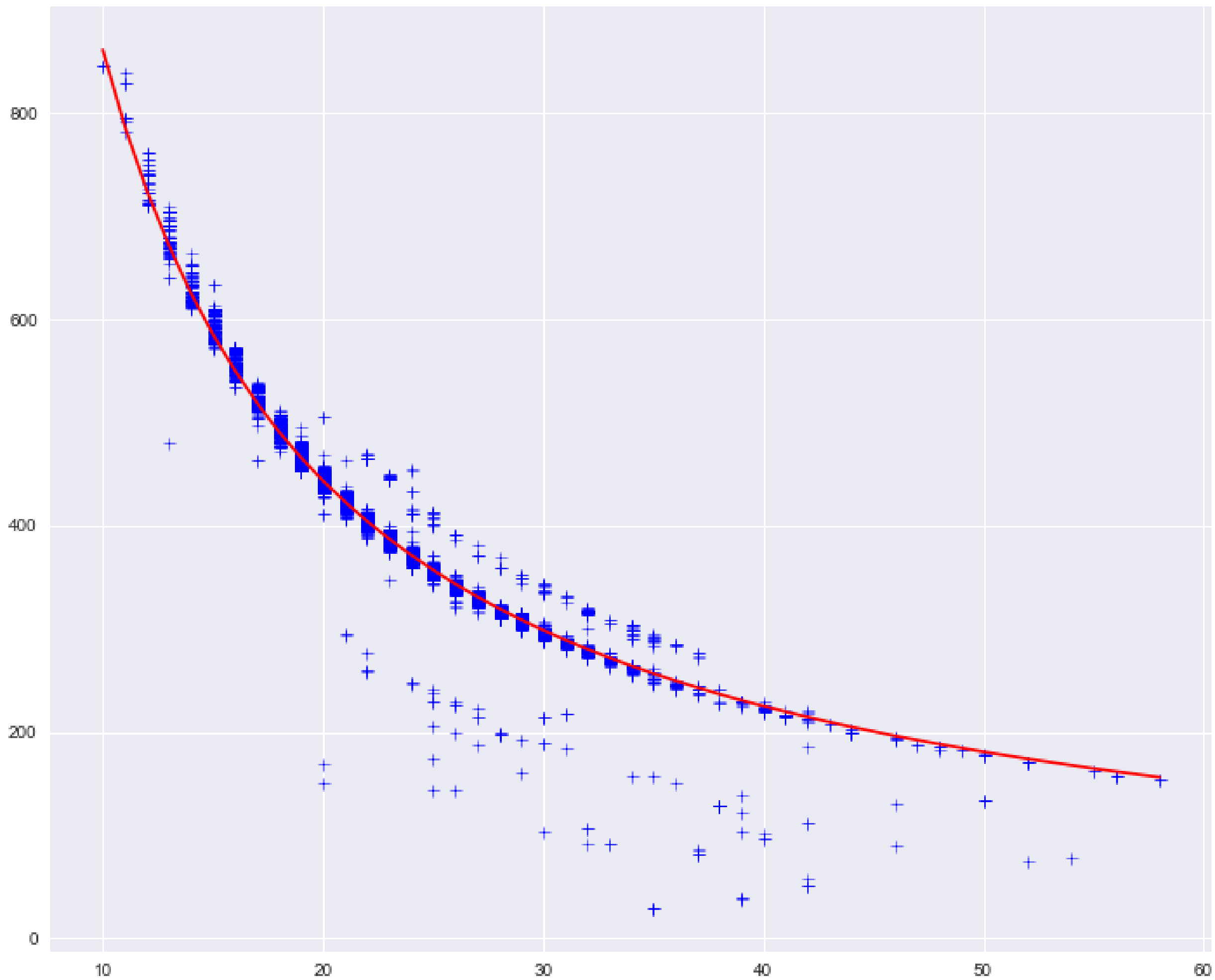
We can calculate the predicted values from our model using the following code. We use *sort_values()* to ensure that the values of the independent variable are in order. This will be important when we plot the curve; we didn't need to do this previously as we relied on the *abline_plot()* function, which handled this for us,

In [42]:
```
prediction = 1/(res.params.const +
                res.params.comb08 * epa_subset.comb08.sort_values())
```

We can now plot the model curve against our original data.

In [43]:
```
fig, axes = plt.subplots(figsize=(12,10))
axes.plot(epa_subset.comb08, epa_subset.co2, 'b+')
axes.plot(epa_subset.comb08.sort_values(), prediction, 'r')
```

Out[43]: [<matplotlib.lines.Line2D at 0x1108efe80>]



Logistic Regression

A [logistic regression (https://en.wikipedia.org/wiki/Logistic_regression)](https://en.wikipedia.org/wiki/Logistic_regression) is used to model data where the dependent variable is categorical. In the simplest case, the dependent variable is binary and has only two possible values. A logistic model, provides an estimate of the probability that one of the two categories applies given the values of the independent variables. We'll only look at simple case where the dependent variable is binary and there is only one independent variable.

An Example

As an example, consider the the following example taken from the Wikipedia page on logistic regressions (https://en.wikipedia.org/wiki/Logistic_regression#Example:_Probability_of_passing_an_exam_versus_
We have two variables: *hours* and *passed*. The *hours* variable represents the number of hours a

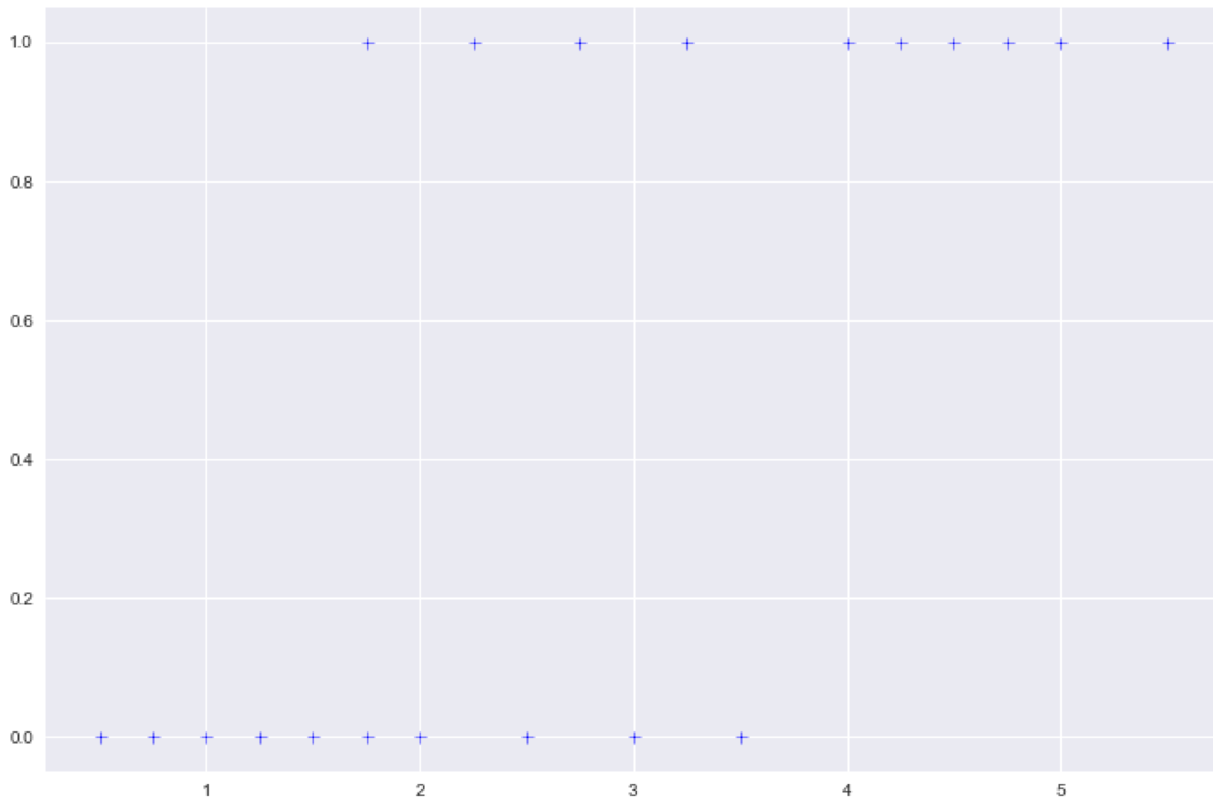
student spent studying for an exam and *passed* indicates whether or not the student passed the exam where `0` indicates failure and `1` indicates that the student passed; *hours* is a continuous variable and *passed* is a discrete, binary variable.

In [44]:
```
hours = [0.5, 0.75, 1.0, 1.25, 1.50, 1.75, 1.75, 2.0, 2.25, 2.5, 2.75, 3.0,
passed = [0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 1, 1, 1]
```

Plotting this data as a scatter plot give the following.

In [45]:
```
plt.plot(hours, passed, 'b+')
```

Out[45]:
```
[<matplotlib.lines.Line2D at 0x110980080>]
```



The logistic regression calculates values for β_0 and β_1 in the following formula

$$P = \frac{1}{1+e^{-(\beta_0+\beta_1 X)}}$$

where P is is a probability value between 0 and 1 and X is the independent variable.

We can use the StatsModels *Logit()* (http://www.statsmodels.org/dev/generated/statsmodels.discrete.discrete_model.Logit.html) function to perform the logistic regression.

We have to reassign the value of `stats.chisqprob` due to a discrepancy between the StatsModels module and the libraries on which it depends.

In [46]:
```
from scipy import stats
stats.chisqprob = lambda chisq, df: stats.chi2.sf(chisq, df)

X = sm.add_constant(hours)
logit_model=sm.Logit(passed,X)
result=logit_model.fit()
display(result.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.401494
         Iterations 7
```

Logit Regression Results

Dep. Variable:	y	**No. Observations:**	20
Model:	Logit	**Df Residuals:**	18
Method:	MLE	**Df Model:**	1
Date:	Wed, 04 Apr 2018	**Pseudo R-squ.:**	0.4208
Time:	22:44:03	**Log-Likelihood:**	-8.0299
converged:	True	**LL-Null:**	-13.863
		LLR p-value:	0.0006365

	coef	std err	z	P>|z|	[0.025	0.975]
const	-4.0777	1.761	-2.316	0.021	-7.529	-0.626
x1	1.5046	0.629	2.393	0.017	0.272	2.737

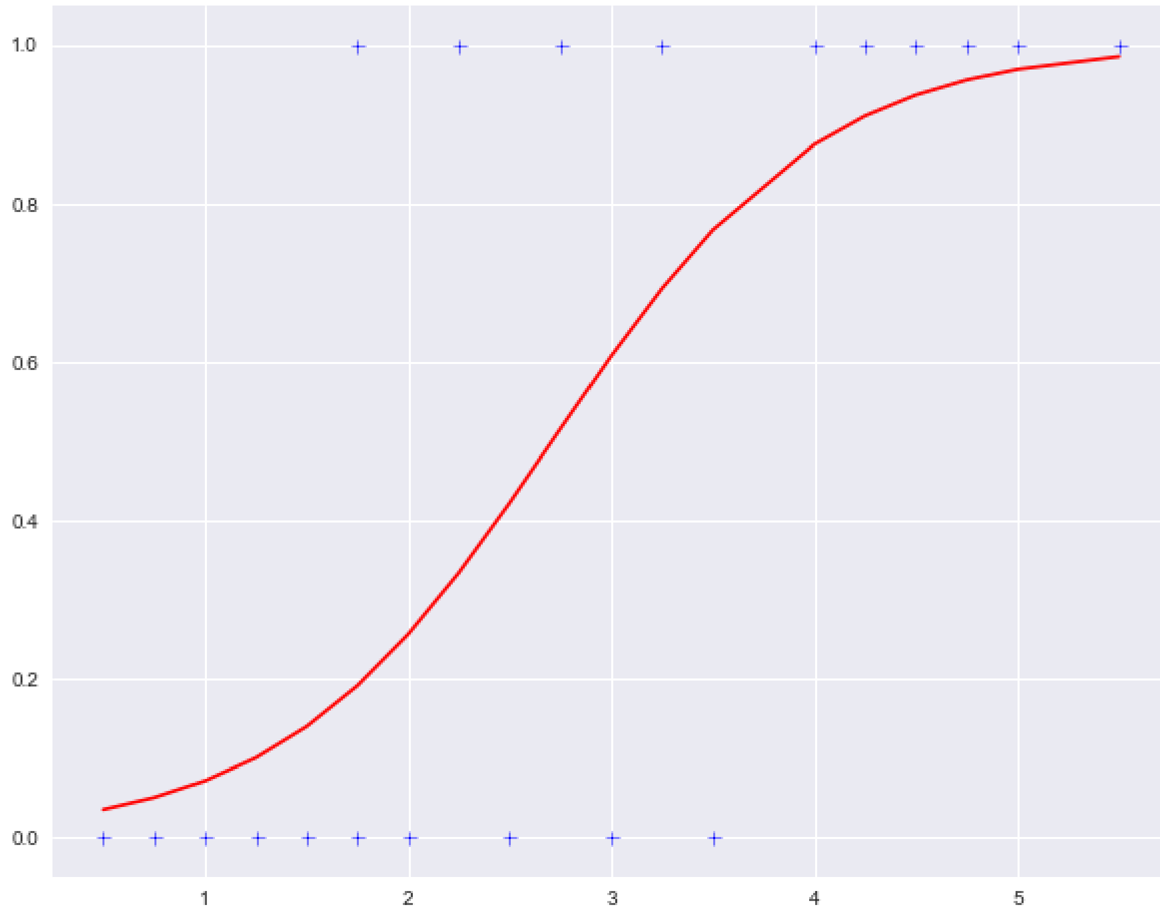
We create the model in much the same way as we did for a linear model. We specify the independent variable and add a constant using StatsModels' *add_constant()* function. We next create a logistic model using the *Logit()* function. To calculate the coefficients, we use model's *fit()* method. After the calculation is complete, we can view a summary of the results.

Just like the linear model, the results of the logistic model have a *predict()* method that give the model's predicted values based on the values of the independent variable. We can use this to plot the logistic curve along with the scatter plot of the data.

The model represents the probability of passing the exam given some number of hours spent studying.

In [47]:
```
fig, axes = plt.subplots(figsize=(10,8))
axes.plot(hours,passed, 'b+')
axes.plot(hours, result.predict(), 'r-')
```

Out[47]:
```
[<matplotlib.lines.Line2D at 0x110a08048>]
```



Home Data

For an example with real data, consider the count auditor data we worked with previously. We can load the data from our local database.

In [48]:
```
from sqlalchemy import create_engine
engine = create_engine('sqlite:///data/output.sqlite')
home_data = pd.read_sql("home_data", con=engine)
home_data.head()
```

Out[48]:

	index	AirConditioning	AppraisedBuilding	AppraisedLand	Area	Bathrooms	Bedrooms	County
0	0	True	59600.0	8100.0	2264	2.0	4.0	Franklin
1	1	True	69800.0	4600.0	1835	1.5	4.0	Franklin
2	2	True	60600.0	4900.0	1656	1.0	3.0	Franklin
3	3	True	31200.0	5000.0	1000	1.0	2.0	Franklin
4	4	True	63300.0	4600.0	1306	2.0	4.0	Franklin

For this example, lets see if we can calculate the logistic model that gives the probability of a house having a fireplace given its area. Currently, the dataset includes the the number of fireplaces a given property has; we need to convert values greater than zero to 1, indicating that there is a fireplace.

First, we drop any rows with missing data. Next, we create a new column, `HasFireplace` that is equal to the mask corresponding the `Fireplaces` being greater than zero. Masks return values of `True` or `False` and the *astype(int)* function call will convert the boolean value to an integer where `False` becomes 0 and `True` becomes 1.

In [49]:
```
home_data.dropna(inplace=True)
home_data['HasFireplace'] = (home_data.Fireplaces > 0).astype(int)
home_data.head()
```

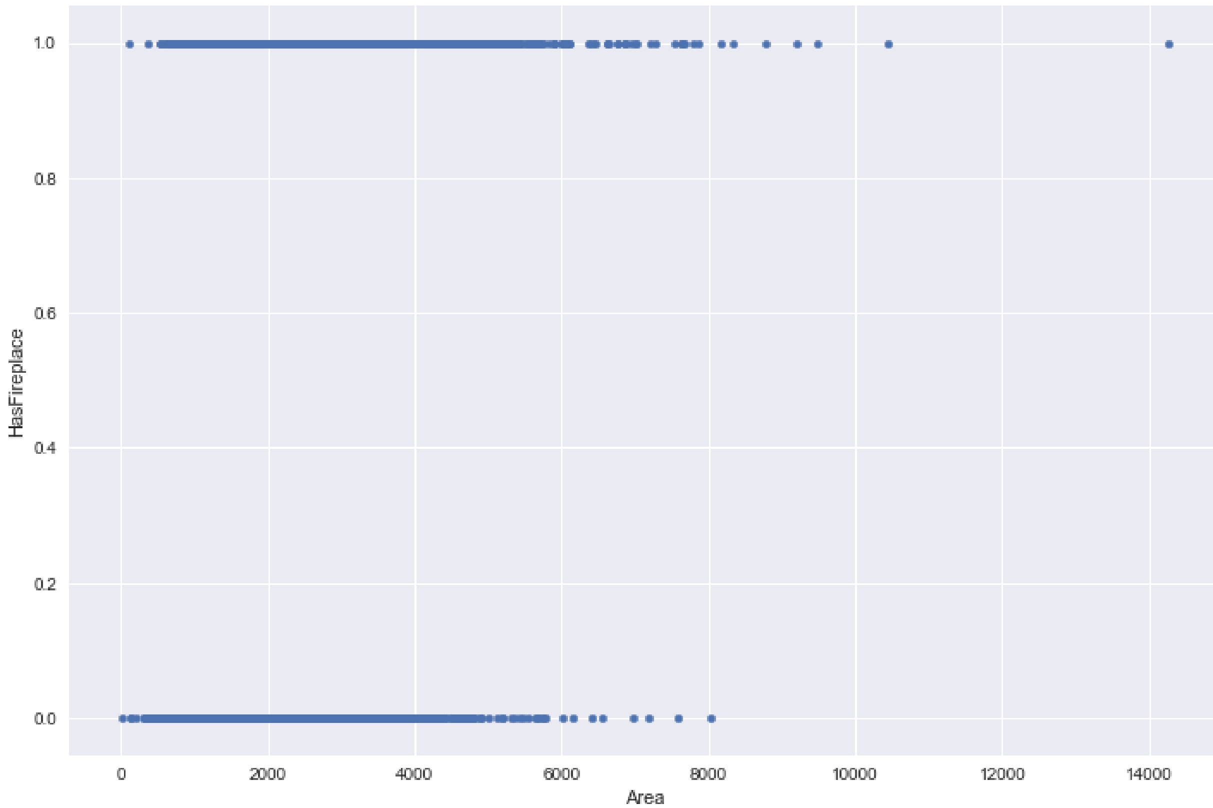
Out[49]:

	index	AirConditioning	AppraisedBuilding	AppraisedLand	Area	Bathrooms	Bedrooms	County
0	0	True	59600.0	8100.0	2264	2.0	4.0	Franklin
1	1	True	69800.0	4600.0	1835	1.5	4.0	Franklin
2	2	True	60600.0	4900.0	1656	1.0	3.0	Franklin
3	3	True	31200.0	5000.0	1000	1.0	2.0	Franklin
4	4	True	63300.0	4600.0	1306	2.0	4.0	Franklin

We can now create the scatter plot of `Area` and `HasFireplaces

In [50]:
```
home_data.plot.scatter(x="Area", y="HasFireplace")
```

Out[50]: <matplotlib.axes._subplots.AxesSubplot at 0x1116cee10>



Before creating the model, we sort the values of our independent variable; this will aid in plotting later.

In [51]:
```
home_data.sort_values(by=["Area"], inplace=True)
```

Creating the logistic model and calculating the regression coefficients is similar to the logistic model as we noted in the example.

In [52]:
```
X = sm.add_constant(home_data.Area)
logit_model=sm.Logit(home_data.HasFireplace,X)
result=logit_model.fit()
display(result.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.636377
         Iterations 5
```

Logit Regression Results

Dep. Variable:	HasFireplace	**No. Observations:**	39236
Model:	Logit	**Df Residuals:**	39234
Method:	MLE	**Df Model:**	1
Date:	Wed, 04 Apr 2018	**Pseudo R-squ.:**	0.05027
Time:	22:44:12	**Log-Likelihood:**	-24969.
converged:	True	**LL-Null:**	-26291.
		LLR p-value:	0.000

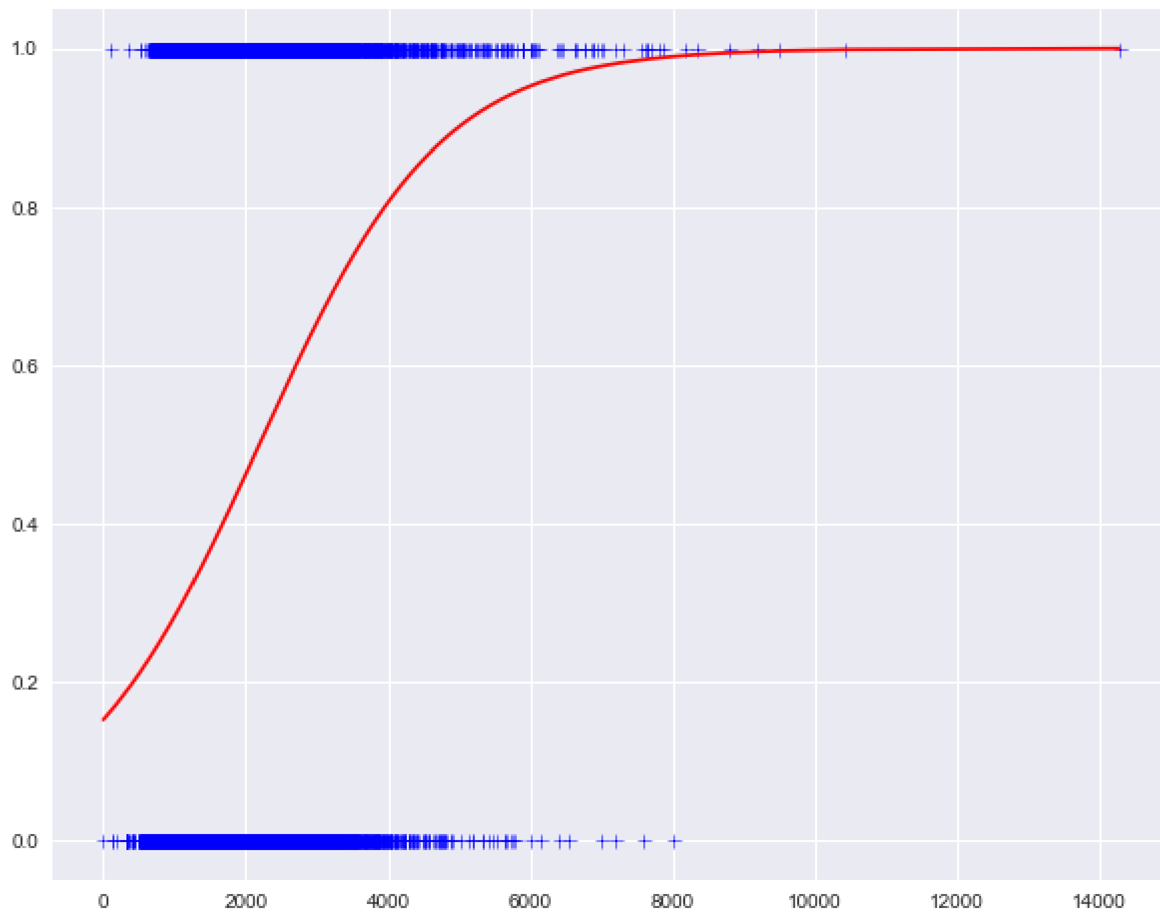
	coef	**std err**	**z**	**P>|z|**	**[0.025**	**0.975]**
const	-1.7215	0.029	-59.697	0.000	-1.778	-1.665
Area	0.0008	1.64e-05	47.992	0.000	0.001	0.001

With the model created and the coefficients calculated, we can now plot the regression curve against the original data.

Lab 7 In the cell below, create a scatter plot of the `Area` and `HasFireplace` columns from the `home_data` DataFrame. On the same figure, plot the logistic regression curve.

In [53]:
```
fig, axes = plt.subplots(figsize=(10,8))
axes.plot(home_data.Area, home_data.HasFireplace, 'b+')
axes.plot(home_data.Area, result.predict(), 'r-')
```

Out[53]: [<matplotlib.lines.Line2D at 0x1118e2940>]



The formula for the curve is given by

$$P = \frac{1}{1+e^{-(-1.7215+0.0008X)}}$$

For a given area, we can calculate the probability that the house has a fireplace using this formula based on the model.

Let's look at one more example. Older home tend to have fewer bathroms. We can create a new column, `MoreThanOneBathroom`

Lab 8 In the cell below, create a new column named `MoreThanOneBathroom` in the `home_data` DataFrame that has a value of 0 if the value of `Bathrooms` is less than or equal to 1 and has a value of 1 otherwise.

In [54]:
```
home_data["MoreThanOneBathroom"] = (home_data.Bathrooms > 1).astype(int)
```

We should sort the data by `YearBuilt` before continuing.

In [55]:
```
home_data.sort_values(by=["YearBuilt"], inplace=True)
```

As before we construct the model with `YearBuilt` as the independent variable and `MoreThanOneBathroom` as the dependent variable. We calculate a logistic curve that best fits the data.

In [56]:
```
X = sm.add_constant(home_data.YearBuilt)
logit_model=sm.Logit(home_data.MoreThanOneBathroom, X)
result=logit_model.fit()
display(result.summary())
```

```
Optimization terminated successfully.
         Current function value: 0.490881
         Iterations 6
```

Logit Regression Results

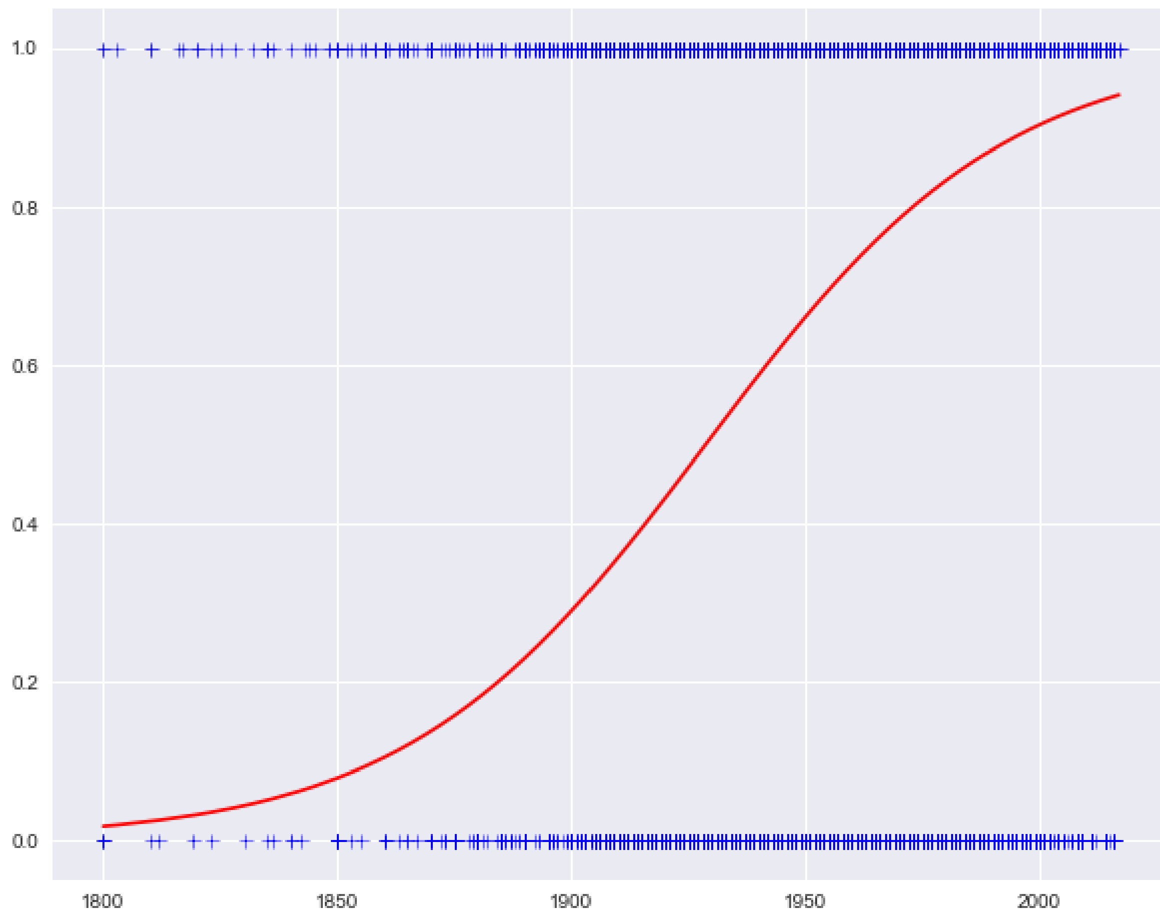
Dep. Variable:	MoreThanOneBathroom	**No. Observations:**	39236
Model:	Logit	**Df Residuals:**	39234
Method:	MLE	**Df Model:**	1
Date:	Wed, 04 Apr 2018	**Pseudo R-squ.:**	0.1333
Time:	22:44:17	**Log-Likelihood:**	-19260.
converged:	True	**LL-Null:**	-22222.
		LLR p-value:	0.000

	coef	**std err**	**z**	**P>|z|**	**[0.025**	**0.975]**
const	-60.3612	0.872	-69.209	0.000	-62.071	-58.652
YearBuilt	0.0313	0.000	70.319	0.000	0.030	0.032

Finally, we can create a scatter plot of `YearBuilt` and `MoreThanOneBathroom` along with the logistic regression curve.

```
In [57]: fig, axes = plt.subplots(figsize=(10,8))
         axes.plot(home_data.YearBuilt, home_data.MoreThanOneBathroom, 'b+')
         axes.plot(home_data.YearBuilt, result.predict(), 'r-')

Out[57]: [&lt;matplotlib.lines.Line2D at 0x111988588&gt;]
```



Lab Answers

1. `sns.pairplot(data=epa_subset[['co2', 'comb08', 'cylinders', 'displ']])`

2. `epa_subset.plot.scatter(x='cylinders', y='displ')`

3. `epa_subset.cylinders.plot(kind='box')`

   and

   `epa_subset.displ.plot(kind='box')`

4. ```
   X = sm.add_constant(epa_no_outliers.cylinders)
   Y = epa_no_outliers.displ
   model = sm.OLS(Y, X)
   res_no_outliers = model.fit()
   ```

5. `epa_non_linear.plot.scatter(x="comb08", y="reciprocal_co2")`
   ```

6.
```
epa_non_linear = remove_outliers(epa_non_linear, "comb08")
epa_non_linear = remove_outliers(epa_non_linear, "reciprocal_co
2")
```

7.
```
fig, axes = plt.subplots(figsize=(10,8))
axes.plot(home_data.Area, home_data.HasFireplace, 'b+')
axes.plot(home_data.Area, result.predict(), 'r-')
```

8.
```
home_data["MoreThanOneBathroom"] = (home_data.Bathrooms > 1).ast
ype(int)
```

Next Steps

The models we create can be used populate dashboards or included in reports. Later, we'll look at creating visualizations and reporting information. Model creation could also be an intermediate step in the analysis process; in a later unit we'll look at automating model creation.

Resources and Further Reading

- Simple and Multiple Linear Regression in Python (https://towardsdatascience.com/simple-and-multiple-linear-regression-in-python-c928425168f9)
- Logistic Regression in Python Using Rodeo (http://blog.yhat.com/posts/logistic-regression-python-rodeo.html)
- Regression Analysis with Python by Massaron and Boschetti (Safari Books) (http://proquest.safaribooksonline.com.cscc.ohionet.org/book/programming/python/9781785286
- Data Science Algorithms in a Week by Natingga, Regression (Safari Books) (http://proquest.safaribooksonline.com.cscc.ohionet.org/book/programming/machine-learning/9781787284586/regression/1500cb6b_9703_4b4a_bffb_61da8fbd2e97_xhtml?uicode=ohlink)

Notes

1. The null hypothesis being tested is that the variable associated with the coefficient has no effect on the dependent variable. When the p-value is sufficiently low, typically less the 0.05, we reject the null hypothesis thereby accepting that the variable does have an effect on the dependent variable.

Exercises

1. Calculate the coefficients for a linear model relating two columns from the fuel economy or county auditor data not discussed in the examples. Create a scatter plot of the data along with a plot of the regression line.

2. Calculate the coefficients for a logistic model relating two columns from the fuel economy or county auditor data not discussed in the examples. Create a scatter plot of the data along with a plot of the regression curve.

In [ ]:

```

```