class4-multiple-regression

September 16, 2014

1 Imports and RNG seed

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

```
In []: import math
       import numpy as np
       import pandas as pd
       from scipy import stats

       from matplotlib import pyplot as plt
```

```
In []: ## Import StatsModel modules we'll need
       import statsmodels.api as sm

       # the formula API allow us to write R-like formulas for models
       import statsmodels.formula.api as smf
```

```
In []: # np.random.seed seeds the random number generator
       # giving a specific seeds allows us to generate pseudo-random numbers
       # deterministically so that our results are reproducible

       np.random.seed(20140906)
```

2 Trees data set

To illustrate multiple regression in R we'll use a built in dataset called `trees.csv`. `trees` consists of measurements of the girth, height, and volume of 31 black cherry trees. We'll start with some summary tables and diagnostic plots to familiarize ourselves with the data:

```
In []: trees = pd.read_csv("https://github.com/pmagwene/Bio723/raw/master/datasets/trees.csv")
```

```
In []: trees.columns
```

```
In []: trees.describe()
```

As one might expect the correlation matrix and correspdong scatterplot matrix shows that all the variables are positively correlated, and girth and volume have a particularly strong correlation.

```
In []: trees.corr()
```

```
In []: from pandas.tools.plotting import scatter_matrix
       scatter_matrix(trees, figsize=(6,6))
       None
```

3 Generating a 3D Scatter Plot

Above we generated a matrix of scatter plots to look at the various pairwise relationships in the data. 3D plots can also be useful too, though one should that such plots are just 2D projections that provide additional information through color and/or scaling of objects.

Matplotlib is capable of producing a variety of 3D plots. See http://matplotlib.org/mpl_toolkits/mplot3d/tutorial.html for an overview.

```
In []: # import the 3D drawing toolkit from matplotlib
       from mpl_toolkits.mplot3d import Axes3D
```

```
In []: fig = plt.figure(figsize=(6,5))  # use figsize to change size and aspect ratio of plot
       ax = fig.add_subplot(111, projection='3d')
       ax.scatter(trees.Girth, trees.Height, trees.Volume)
       ax.set_xlabel('Girth')
       ax.set_ylabel('Height')
       ax.set_zlabel('Volume')
```

3.1 Changing 3D plot orientation

As mentioned above, 3D plots are 2D projections corresponding to different observer locations with respect to the data being plotted. To facilitate exploration we can change our orientation with respect to the data.

3D plotting and graphics libraries often use spherical coordinate systems to describe the viewers posistion relative to an object of interest. The axes in this spherical coordinate system are often referred to as elevation, azimuth, and distance and are illustrated in the figure below, found at the MathWorks website. For more info, see http://www.mathworks.com/help/matlab/visualize/setting-the-viewpoint-with-azimuth-and-elevation.html.

```
In []: # illustrating the spherical coordinate system for 3D plots

       from IPython.display import display, Image
       img = Image(url="http://www.mathworks.com/help/matlab/visualize/chview3.gif")
       display(img)
```

```
In []: # change orientation of plot using azimuth, orientation, and distance arguments
       # elev = elevation along z axis in degrees (default = 30)
       # azim = orientation in x,y plane in degrees (default = -60)
       # dist = distance of viewingpoint from object(defualt = 10)

       fig = plt.figure(figsize=(6,5))
       ax = fig.add_subplot(111, projection='3d', azim=-30, elev = 45)
       ax.scatter(trees.Girth, trees.Height, trees.Volume)
       ax.set_xlabel('Girth')
       ax.set_ylabel('Height')
       ax.set_zlabel('Volume')
```

4 Interactive Plots with IPython Widgets

For purposes of data exploration, changing coordinates by hand can be tedious. Luckily, recent versions of the IPython notebook support a set of "interactive widgets" that allow us to interactively explore our data. This is a relatively new feature of IPython, and thus is not particularly well documented, but some information can be found here: IPython Widget Examples.

Using the basic interactive widgets with a plot require two steps:

1. Wrap your plotting code into a function that takes arguments that change the plot

2. Pass the new plotting function to the `interact()` function and specify the allowable ranges for the arguments to the plotting function.

Below we illustrate how to interactively change the azimuth and elevation of a 3D plot using the IPython interact widgets. Use the corresponding sliders to change the plot.

```
In []: def interactive_scatter(azim = -60, elev = 30):
            fig = plt.figure(figsize=(6,5))
            ax = fig.add_subplot(111, projection='3d', azim=azim, elev = elev)
            ax.scatter(trees.Girth, trees.Height, trees.Volume)
            ax.set_xlabel('Girth')
            ax.set_ylabel('Height')
            ax.set_zlabel('Volume')


       from IPython.html.widgets import interact

       i = interact(interactive_scatter,
               azim = (-180,180, 5),  # min, max, step associated with each variable
               elev = (-90,90, 5),
               )
```

5 Multiple Regression

Let's assume we're lumberjacks, but our permit only allows us to harvest a fixed number of trees. We get paid by the total volume of wood we harvest, so we're interested in predicting a tree's volume (hard to measure directly) as a function of its girth and height (relatively easy to measure), so we can pick the best trees to harvest.

From the 3D scatter plot it looks like we ought to be able to find a plane through the data that fits the scatter fairly well. We'll therefore calculate a multiple regression of volume on height and width.

The Ordinary Least Squares functions of the `StatsModels` package, that we first explored last week, can be used to calculate a multiple regression. As we did last week, we'll use R-style formulas to specify our model.

```
In []: # specify and fit model
       treefit = smf.ols('Volume ~ Girth + Height', data = trees).fit()
```

```
In []: print treefit.summary()
```

The regression equation is: $\hat{y} = 4.71x_1 + 0.34x_2$, where y is Volume, and x_1 and x_2 are Girth and Height respectively. Since they're on different scales the coefficients for Girth and Height aren't directly comparable. Both coefficients are significant at the $p < 0.05$ level, but note that Girth is the much stronger predictor. In fact the addition of height explains only a minor additional fraction of variation in tree volume, so from the lumberjack's perspective the additional trouble of measuring height probably isn't worth it.

6 Visualizing the regression in variable space

To visualize the multiple regression, let's draw the 3D scatter of points and the plane that corresponds to the regression model:

```
In []: # get coefficients of fit
       b0 = treefit.params.Girth
       b1 = treefit.params.Height
       a = treefit.params.Intercept
```

```
# generate a grid of points over the ranges represented by the data
gmin, gmax = trees.Girth.min(), trees.Girth.max()
hmin, hmax = trees.Height.min(), trees.Height.max()
X, Y = np.meshgrid(np.linspace(gmin, gmax, 50), np.linspace(hmin, hmax, 50))

# fit the model to the grid of points
Z = a + b0*X + b1*Y
```

```
In []: def interactive_scatter2(azim = -60, elev = 30):
           fig = plt.figure(figsize=(6,5))
           ax = fig.add_subplot(111, projection='3d', azim=azim, elev = elev)
           ax.plot_surface(X, Y, Z, color='grey', alpha=0.2)
           ax.scatter(trees.Girth, trees.Height, trees.Volume, color='red')
           ax.set_xlabel('Girth')
           ax.set_ylabel('Height')
           ax.set_zlabel('Volume')

       i = interact(interactive_scatter2,
               azim = (-180,180, 5),  # min, max, step associated with each variable
               elev = (-90,90, 5),
               )
```

From the figure it looks like the regression model fits pretty well, as we anticipated from the pairwise relationships.

7 Exploring the Vector Geometry of the Regression Model

Let's use our knowledge of vector geometry to further explore the relationship between the predicted Volume and the predictor variables.

By definition the vector representing the predicted values lies in the plane defined by Height and Girth, so let's do some simple calculations to understand their length and angular relationships.

7.0.1 Vector lengths and angular relationshpis

Recall that in our vector representation, the length of the vectors is proportional to the standard deviation of the variables they represent, and the angular relationships between the vectors is related to their correlation.

```
In []: np.std(trees.Height, ddof=1)
```

```
In []: np.std(trees.Girth, ddof=1)
```

```
In []: np.std(treefit.fittedvalues, ddof=1)
```

```
In []: correlations = np.corrcoef([trees.Height, trees.Girth, treefit.fittedvalues])
       correlations
```

```
In []: # angles btw vectors in degress
       np.arccos(correlations) * (180/math.pi)
```

7.1 In-class assignment

Using the calculations above you should now be able to sketch out by hand, a diagram depicting the vector relationships between Height, Girth, and the predicted Volume . Once you've finished with your sketch, discuss it with your fellow classmates. Did you get similar answers? If not, discuss it and try to come up with an agreed upon representation.

7.2 Exploring the Residuals from the Regression

Now let's look at the residuals from the regression. The residuals represent the 'unexplained' variance:

```
In []: xmin, xmax = 0, 90
       plt.plot(trees.Volume, treefit.resid, 'ko')
       plt.hlines(0, xmin, xmax,linestyle='dashed',color='r', linewidth=1.5)
       plt.xlim(xmin, xmax)
```

Ideally the residuals should be evenly scattered around zero, with no trends as we go from high to low values of the dependent variable. As you can see in the figure above it looks like that the residuals on the left tend to be below zero, while those on the far right of the plot are consistently above zero, suggesting that there may be a non-linear aspect of the relationship that our model isn't capturing.

Let's think about the relationships we're actually modeling for a few minutes. For the sake of simplicity let's consider the trunk of a tree to be a cylinder. How do the dimensions of this cylinder relate to its volume? You can look up the formula for the volume of a cylinder, but the key thing you'll want to note is that volume of the cylinder should be proportional to a characteristic length of the cylinder cubed ($V \propto \mathrm{L}^3$). This suggests that if we want to fit a linear model we should relate Girth to $\sqrt[3]{\mathrm{Volume}}$. Let's explore this a little. Since our initial multiple regression suggested that height had relatively little predictive power, we'll simplify our model down to a single predictor:

```
In []: cuberootvol = np.power(trees.Volume, 0.33)
```

```
In []: np.corrcoef([trees.Volume, cuberootvol, trees.Girth])
```

```
In []: lm_orig = smf.ols("Volume ~ Girth", data = trees).fit()
```

```
In []: trans_data = pd.DataFrame({"Girth":trees.Girth,
                                  "CubeRootVol":cuberootvol})

       lm_trans = smf.ols("CubeRootVol ~ Girth", data = trans_data).fit()
```

```
In []: print lm_orig.summary()
```

```
In []: print lm_trans.summary()
```

Comparing the summary tables, we see indeed that using the cube root of Volume improves the fit of our model some. Let's examine the residuals.

```
In []: # generate a figure with 2 subplots
       # first argument is number of rows, second is number of columns
       fig, (ax1, ax2) = plt.subplots(2, 1)

       xmin, xmax = 0, 90

       ax1.plot(trees.Volume, lm_orig.resid, 'ko')
       ax1.hlines(0, xmin, xmax, linestyle='dashed', color='r')
       ax1.set_ylabel('Residuals')
       ax1.set_xlabel('Volume')

       ax2.plot(trees.Volume, lm_trans.resid, 'ko')
       ax2.hlines(0, xmin, xmax, linestyle='dashed', color='r')
       ax2.set_ylabel('Residuals')
       ax2.set_xlabel('$\sqrt[3]{\mathrm{Volume}}$')

       plt.subplots_adjust(hspace = 0.5)
```

As we can see the transformation we applied to the data did seem to make our residuals more uniform across the range of observations.

8 Fitting a Curvilinear Model

Above we transformed the volume data in order to fit a straight line relationship between $\sqrt[3]{V}$ and Girth. However, we could just as easily have applied a cubic regression to the original variables as shown below. Remember this model is still linear in the coefficients:

```
In []: tree3 = smf.ols("Volume ~ I(Girth**3)", data = trees).fit()
```

The `I()` syntax used above requires a little explanation. The `I()` function "protects" the object in it's argument; in this case telling the regression function to treat this as Girth raised to the third power as opposed to trying to construct interaction terms for Girth.

```
In []: print tree3.summary()
```

```
In []: a = tree3.params[0]
       b0 = tree3.params[1]

       # fit the curvilinear model over a range of girths
       x = np.linspace(8, 22, 100)
       fit = a + b0 * x**3
```

```
In []: plt.plot(trees.Girth, trees.Volume, 'ko')
       plt.plot(x, fit, 'r')
       plt.xlim(8,22)
       plt.ylim(0,100)
       plt.xlabel('Girth')
       plt.ylabel('Volume')

       textstr = "Volume = {:0.3f} + {:0.3f} Girth$^3$".format(a,b0)
       plt.text(10, 80, textstr)
```

9 Exploring the impact of nearly collinear predictors on regression

In lecture we discussed the problems that can arise in regression when your predictor variables are nearly collinear. In this section we'll illustrate some of these issues.

Consider again the `trees` data set. Recall that two of the variables – Girth and Volume – are highly correlated and thus nearly collinear. Let's explore what happens when we treat Height as the dependent variable, and Girth and Volume as the predictor variables.

```
In []: lm_H = smf.ols('Height ~ Girth + Volume', data = trees).fit()
       print lm_H.summary()
```

Now, let's created a slightly different version of the trees data set by adding some noise to the three variables. Our goal here is to simulate a data set we might have created had we measured a slightly different set of trees during our sampling.

```
In []: noisy = pd.DataFrame()

       noisy['Girth']= trees.Girth + np.random.normal(scale = 0.25*trees.Girth.std(),
                                                      size=len(trees))

       noisy['Height'] = trees.Height + np.random.normal(scale = 0.25*trees.Height.std(),
                                                         size=len(trees))

       noisy['Volume']= trees.Volume + np.random.normal(scale = 0.25*trees.Volume.std(),
                                                        size=len(trees))
```

Here we added normally distribution noise to the data set. The noise has a standard deviation equivalent to one-quarter the standard deviation of the original variables. Let's take a moment to convince ourselves that our new data set, `noisy`, is not too different from the `trees` data set from which it was derived.

```
In []: print noisy.describe()
```

```
In []: print noisy.corr()
```

```
In []: np.corrcoef(noisy.Girth, trees.Girth)
```

```
In []: np.corrcoef(noisy.Volume, trees.Volume)
```

```
In []: np.corrcoef(noisy.Height, trees.Height)
```

Now that we've convinced ourselves that our jittered data set is a decent approximation to our original data set, let's re-calculate the linear regression, and compare the coefficients of the jittered model to the original model:

```
In []: lm_Hnoisy = smf.ols('Height ~ Girth + Volume', data = noisy).fit()
```

```
In []: print lm_Hnoisy.params
```

```
In []: print lm_H.params
```

We see that the coefficients of the linear model have changed quite a bit between the original data and the noisy data. Our model is unstable to relatively modest changes to the data!

Let's draw some plots to illustrate how different the models fit to the original and noisy data are:

```
In []: # get coefficients of fit for original and noisy data
       b0 = lm_H.params.Girth
       b1 = lm_H.params.Volume
       a = lm_H.params.Intercept

       b0_n = lm_Hnoisy.params.Girth
       b1_n = lm_Hnoisy.params.Volume
       a_n = lm_Hnoisy.params.Intercept

       # generate a grid of points over the ranges represented by the data
       gmin, gmax = trees.Girth.min(), trees.Girth.max()
       vmin, vmax = trees.Volume.min(), trees.Volume.max()
       X, Y = np.meshgrid(np.linspace(gmin, gmax, 50), np.linspace(vmin, vmax, 50))

       # fit the  models to the grid of points
       Zorig = a + b0*X + b1*Y
       Znoisy = a_n + b0_n*X + b1_n*Y

       def interactive_scatter3(azim = -60, elev = 30):
           fig = plt.figure()
           ax = fig.add_subplot(111, projection='3d', azim=azim, elev = elev)
           ax.plot_surface(X, Y, Zorig, color='grey', alpha=0.2)
           ax.plot_surface(X, Y, Znoisy, color='red', alpha=0.2)
           ax.set_xlabel('Girth')
           ax.set_ylabel('Volume')
           ax.set_zlabel('Hieght')

       i = interact(interactive_scatter3,
               azim = (-180,180, 5),  # min, max, step associated with each variable
               elev = (-90,90, 5),
           )
```

9.1 Comparison with non-collinear predictors

Let's do the same comparison for the multiple regression of Volume on Height and Girth, using exactly the same noisy data set as above. In this case the predictor variables are *not* nearly collinear, so we expect to model should be relatively stable.

```
In []: lm_V = smf.ols('Volume ~ Girth + Height', data = trees).fit()
       lm_Vnoisy = smf.ols('Volume ~ Girth + Height', data = noisy).fit()
```

```
In []: print "V ~ Girth + Height, Original data"
       print lm_V.params
       print
       print "V ~ Girth + Height, Noisy data"
       print lm_Vnoisy.params
```

```
In []: # get coefficients of fit for original and noisy data
       b0 = lm_V.params.Girth
       b1 = lm_V.params.Height
       a = lm_V.params.Intercept

       b0_n = lm_Vnoisy.params.Girth
       b1_n = lm_Vnoisy.params.Height
       a_n = lm_Vnoisy.params.Intercept

       # generate a grid of points over the ranges represented by the data
       gmin, gmax = trees.Girth.min(), trees.Girth.max()
       hmin, hmax = trees.Height.min(), trees.Height.max()
       X, Y = np.meshgrid(np.linspace(gmin, gmax, 50), np.linspace(hmin, hmax, 50))

       # fit the  models to the grid of points
       Zorig = a + b0*X + b1*Y
       Znoisy = a_n + b0_n*X + b1_n*Y

       def interactive_scatter4(azim = -60, elev = 30):
           fig = plt.figure()
           ax = fig.add_subplot(111, projection='3d', azim=azim, elev = elev)
           ax.plot_surface(X, Y, Zorig, color='grey', alpha=0.2)
           ax.plot_surface(X, Y, Znoisy, color='red', alpha=0.2)
           ax.set_xlabel('Girth')
           ax.set_ylabel('Height')
           ax.set_zlabel('Volume')

       i = interact(interactive_scatter4,
               azim = (-180,180, 5),  # min, max, step associated with each variable
               elev = (-90,90, 5),
           )
```

As the figure above illustrates, the planes representing the model fits for the original and noisy data are very similar, implying our results are relatively robust to minor variations in the data.

9.2 Vector geometry of the model with colinear predictors

Finally, let's do some vector calculations to quantify how the angular deviation between the fit data and the predictor variables changes between the original and jittered data set for the two different multiple regressions:

```
In []: # write a quickie fxn to express angle between vectors in degrees
       def vec_angle(x,y):
           angmtx =  np.arccos(np.corrcoef(x,y)) * (180/np.pi)
           return angmtx[0,1]

In []: # vector angles for fit of Height ~ Girth + Volume (orig)

       print "Angular deviations (degrees)"
       print "Height ~ Girth + Volume, original data"
       print "=" * 40
       print "fit(Height), Girth: ", vec_angle(lm_H.fittedvalues, trees.Girth)
       print "fit(Height), Volume: ",vec_angle(lm_H.fittedvalues, trees.Volume)

       print "\n"

       print "Angular deviations (degrees)"
       print "Height ~ Girth + Volume, noisy data"
       print "=" * 40
       print "fit(Height), Girth: ", vec_angle(lm_Hnoisy.fittedvalues, trees.Girth)
       print "fit(Height), Volume: ",vec_angle(lm_Hnoisy.fittedvalues, trees.Volume)

In []: # vector angles for fit of Volume ~ Girth + Height

       print "Angular deviations (degrees)"
       print "Volume ~ Girth + Height, original data"
       print "=" * 40
       print "fit(Volume), Girth: ", vec_angle(lm_V.fittedvalues, trees.Girth)
       print "fit(Volume), Height: ",vec_angle(lm_V.fittedvalues, trees.Height)

       print "\n"

       print "Angular deviations (degrees)"
       print "Height ~ Girth + Height, noisy data"
       print "=" * 40
       print "fit(Volume), Girth: ", vec_angle(lm_Vnoisy.fittedvalues, trees.Girth)
       print "fit(Volume), Height: ",vec_angle(lm_Vnoisy.fittedvalues, trees.Height)
```