Monday, 17 August 2015

R, Python, and SAS: Getting Started with Linear Regression

Posted by Al-Ahmadgaid Asaad

Consider the linear regression model,

$$y_i = f_i(\boldsymbol{x}|\boldsymbol{\beta}) + \varepsilon_i,$$

where y_i is the *response* or the *dependent* variable at the ith case, $i = 1, \cdots, N$ and the *predictor* or the *independent* variable is the $\boldsymbol{x}$ term defined in the mean function $f_i(\boldsymbol{x}|\boldsymbol{\beta})$. For simplicity, consider the following simple linear regression (SLR) model,

$$y_i = \beta_0 + \beta_1 x_i + \varepsilon_i.$$

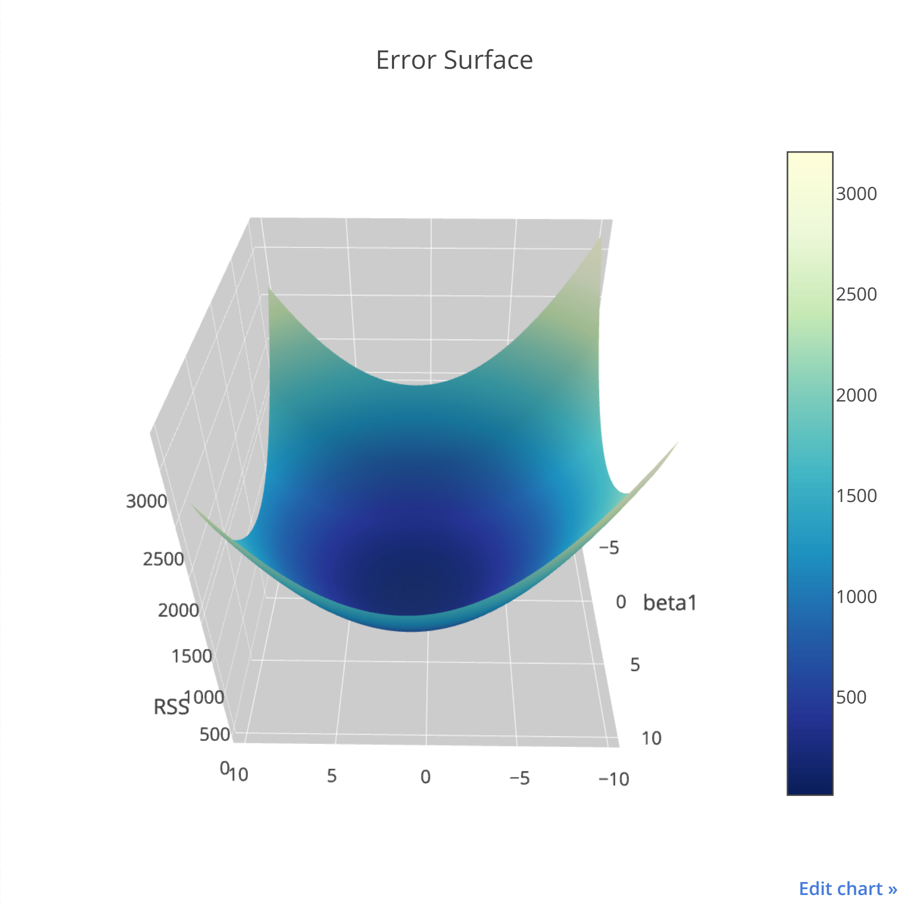
To obtain the (best) estimate of β_0 and β_1, we solve for the least residual sum of squares (RSS) given by,

$$S = \sum_{i=1}^{n} \varepsilon_i^2 = \sum_{i=1}^{n} (y_i - \beta_0 - \beta_1 x_i)^2.$$

Now suppose we want to fit the model to the following data, **Average Heights and Weights for American Women**, where *weight* is the response and *height* is the predictor. The data is available in R by default.

```
data(women)
women

# OUTPUT
   height weight
1      58    115
2      59    117
3      60    120
4      61    123
5      62    126
6      63    129
7      64    132
8      65    135
9      66    139
10     67    142
11     68    146
12     69    150
13     70    154
14     71    159
15     72    164
```

The following is the plot of the residual sum of squares of the data base on the SLR model over β_0 and β_1, note that we standardized the variables first before plotting it,



If you are interested on the codes of the above figure, please click here. To minimize this elliptic paraboloid, differentiation has to be done with respect to the parameters, and then equate this to zero to obtain the stationary point, and finally solve for β_0 and β_1. For more on derivation of the estimates of the parameters see reference 1.

Simple Linear Regression in R

In R, we can fit the model using the `lm` function, which stands for linear models, i.e.

```
library(magrittr)
model <- {weight ~ height} %>% lm(data = women)
```

Formula, defined above as `{response ~ predictor}`, is a handy method for fitting model to the data in R. Mathematically, our model is

$$weight = \beta_0 + \beta_1(height) + \varepsilon.$$

The summary of it is obtain by running `model %>% summary` or for non-magrittr user `summary(model)`, given the `model` object defined in the previous code,

```
model %>% summary

# OUTPUT
Call:
lm(formula = ., data = women)

Residuals:
    Min      1Q  Median      3Q     Max 
-1.7333 -1.1333 -0.3833  0.7417  3.1167 

Coefficients:
             Estimate Std. Error t value Pr(>|t|)    
(Intercept) -87.51667    5.93694  -14.74 1.71e-09 ***
height        3.45000    0.09114   37.85 1.09e-14 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.525 on 13 degrees of freedom
Multiple R-squared:  0.991,     Adjusted R-squared:  0.9903 
F-statistic:  1433 on 1 and 13 DF,  p-value: 1.091e-14
```

The Coefficients section above returns the estimated coefficients of the model, and these are $\beta_0 = -87.51667$ and $\beta_1 = 3.45000$ (it should be clear that we used the unstandardized variables for obtaining these estimates). The estimates are both significant base on the p-value under .05 and even in .01 level of the test. Using the estimated coefficients along with the residual standard error we can now construct the fitted line and it's confidence interval as shown below.

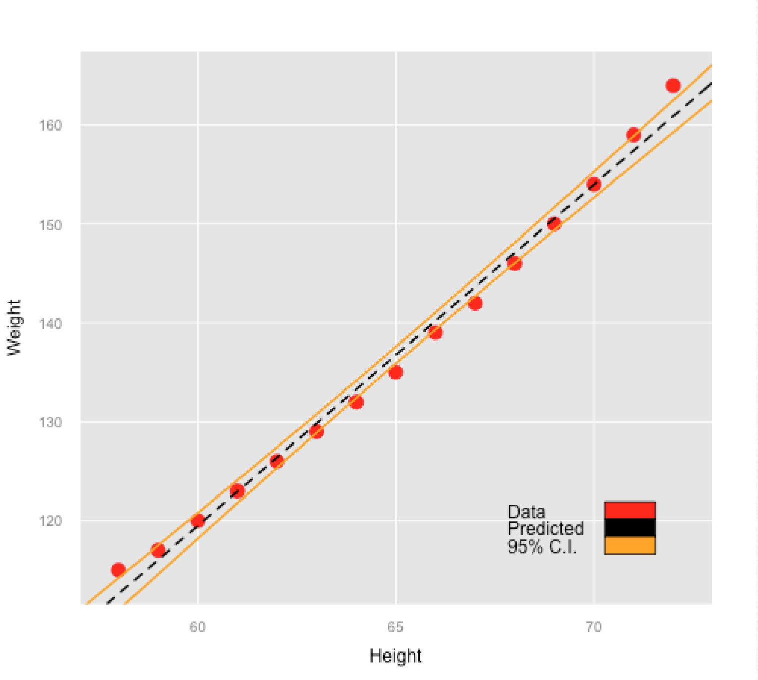


Fig 1. Plot of the Data and the Predicted Values in R.

```
library(lattice)
library(latticeExtra)

{weight ~ height} %>% xyplot(
  data = women, type = c('g', 'p'),
  xlab = 'Height', ylab = 'Weight',
  par.settings = ggplot2like(col = 'red', cex = 1.5),
  panel = function(x, y, ...) {
    panel.xyplot(x, y, ...)
    pred <- function (x, y = 1) {
      p <- model %>%
        predict(newdata = data.frame(height = x), interval = "confidence", level = .95) %>%
        as.data.frame
      p[y] %>% c %>% unlist
    }
    panel.curve(pred(x), lty = 'dashed', lwd = 2)
    panel.curve(pred(x, y = 2), lwd = 2, col = 'orange')
    panel.curve(pred(x, y = 3), lwd = 2, col = 'orange')
  },
  key = list(
    corner = c(.9, .1),
    text = list(label = c('Data', 'Predicted', 'C.I.')),
    rectangle = list(col = c('red', 'black', 'orange'))
  )
)
```

Simple Linear Regression in Python

In Python, there are two modules that have implementation of linear regression modelling, one is in scikit-learn (`sklearn`) and the other is in Statsmodels (`statsmodels`). For example we can model the above data using `sklearn` as follows:

```
from sklearn import linear_model
from pandas import DataFrame

dat = {'height': [58, 59, 60, 61, 62, 63, 64, 65,
                  66, 67, 68, 69, 70, 71, 72],
       'weight': [115, 117, 120, 123, 126, 129, 132,
                  135, 139, 142, 146, 150, 154, 159, 164]}

women = DataFrame(data = dat, columns = ['height', 'weight'])
model = linear_model.LinearRegression(fit_intercept = True)
height = women.height.reshape(len(women), 1)
weight = women.weight.reshape(len(women), 1)
fit = model.fit(height, weight)
```

```
print 'Intercept: %.4f, Height: %.4f' % (fit.intercept_, fit.coef_)

# OUTPUT
Intercept: -87.5167, Height: 3.4500
```

Above output is the estimate of the parameters, to obtain the predicted values and plot these along with the data points like what we did in R, we can wrapped the functions above into a class called `linear_regression` say, that requires Seaborn package for neat plotting, see the codes below:

```
__author__ = 'al-ahmadgaidasaad'

from sklearn import linear_model
from pandas import read_csv, DataFrame
import numpy as np
from scipy.stats import t
import seaborn
import matplotlib.pylab as plt

class linear_regression(object):
    """ Fit linear model to the data.

    Parameters
    ----------
    x : numpy array or sparse matrix of shape [n_samples,n_features]
        Independent variable defined in the column of data argument below.
    y : numpy array of shape [n_samples, n_targets]
        Dependent variable defined in the column of the data argument below.
    data: pandas DataFrame or str instance (local path/directory of the data)
        Data frame with columns x and y defined above.
    intercept: boolean, default False
        Toggle intercept of the model.

    Examples
    --------
    >>> model = linear_regression('height', 'weight', data = 'women.csv')
    >>> print model
    >>> model = linear_regression('height', 'weight', data = 'women.csv', intercept = True)
    >>> print model
    """

    def __init__(self, x, y, data, intercept = False):
        self.intercept = intercept
        self.x = str(x)
        self.y = str(y)

        if isinstance(data, str):
            self.data = read_csv(data)
        else:
            if isinstance(data, DataFrame):
                self.data = data
            else:
                raise TypeError('%s should be a pandas.DataFrame instance' % data)

        self.indv = np.array(self.data.ix[:, x]).reshape((len(self.data), 1))
        self.depv = np.array(self.data.ix[:, y]).reshape((len(self.data), 1))
        _regr_ = linear_model.LinearRegression(fit_intercept = self.intercept)
        self.fit = _regr_.fit(self.indv, self.depv)

    def __str__(self):
        if self.intercept is True:
            _model_ = 'Model:\n' \
                    '\t(%s) = %6.3f + %6.3f * (%s) + error' % (self.y, self.fit.intercept_, self.fit.coef_, self.x)
            _summary_ = 'Summary:\n\t\t\t\tEstimates\n' \
                      '\t(Intercept)\t %8.3f\n' \
                      '\t%s\t\t %8.3f' % (self.fit.intercept_, self.x, self.fit.coef_)
        else:
            _model_ = 'Model:\n' \
                      '\t(%s) = %6.3f * (%s) + error' % (self.y, self.fit.coef_, self.x)
            _summary_ = 'Summary:\n\t\t\tEstimates\n' \
                        '\t%s\t\t %8.3f' % (self.x, self.fit.coef_)
```

```python
        return '%s\n\n%s' % (_model_, _summary_)

    def predict(self, x = None, plot = False, conf_level = 0.95, save_fig = False, filename = 'figure', fig_format = '.pdf'
        """ Predict linear model given x.

        Parameters
        ----------
        x : numpy array or sparse matrix of shape [n_samples,n_features], default None
            Independent variable, if set to None, the original X (predictor) variable
            of the model will be used.
        plot : boolean, default False
            Toggle plot of the data points along with its predicted values and confidence interval.
        conf_level: float between 0 and 1, default 0.95
            Confidence level of the confidence interval in plot. Enabled if plot is True.
        save_fig: boolean, default False
            Toggle to save plot.
        filename: str, default 'figure'
            Name of the file if save_fig is True.
        fig_format: str, default 'pdf'
            Format of the figure if save_fig is True, choices are: 'png', 'ps', 'pdf', and 'svg'.

        Examples
        --------
        >>> from pandas import DataFrame
        >>> from numpy import random.normal
        >>> df = {'x': random.normal(50, 25, 5), 'y': random.normal(50, 25, 5)}
        >>> model = linear_regression('height', 'weight', data = 'women.csv')
        >>> model.predict()

        Returns
        --------
        _res_df_: pandas DataFrame of shape [n_samples,n_features]
            A DataFrame of columns (features) 'Predicted', 'Lower' (Confidence Limit), 'Upper' (Confidence Limit)

        See Also
        --------
        sklearn.linear_model.LinearRegression.predict
            Predict using the linear model
        """
        if x is not None and isinstance(x, np.ndarray) and len(x.shape) is 1:
            _x_ = x.reshape((len(x), 1))
        elif x is not None and isinstance(x, np.ndarray) and len(x.shape) is 2 and x.shape[0] is len(x):
            _x_ = x
        elif x is None:
            _x_ = self.indv
        else:
            raise TypeError('%s should be one dimensional numpy array' % x)

        _yhati_ = self.fit.predict(self.indv)
        _yhat_ = self.fit.predict(_x_)
        _ci_ = self.yhat_ci(_yhat_, _yhati_, _x_, alpha = 1 - conf_level)

        if plot is True:
            plt.plot(self.indv, self.depv, 'o', color = 'red', label = 'Data Points', markersize = 8)
            plt.plot(_x_, _yhat_, '--', color = 'black', label = 'Fitted Values')
            plt.plot(_x_, _ci_[:,0], '-', color = 'orange', label = '%.1f Confidence Interval' % (conf_level * 100))
            plt.plot(_x_, _ci_[:,1], '-', color = 'orange')
            plt.legend(loc = 'lower right')
            if save_fig is True:
                plt.savefig(filename + '.' + fig_format)
            else:
                plt.show

        _res_mat_ = np.column_stack((_yhat_, _ci_))
        _res_df_ = DataFrame(data = {'Predicted':_res_mat_[:,0], 'Lower':_res_mat_[:,1], 'Upper':_res_mat_[:,2]},
                             columns = ['Predicted', 'Lower', 'Upper'])

        return _res_df_

```

```
    def residual_stderror(self, yhat):
        _ysum_ = np.sum((self.depv - yhat) ** 2)
        _sy_ = (_ysum_ * 1.) / (len(self.depv) - 2)
        return np.sqrt(_sy_)

    def yhat_ci(self, yhat, yhati, x, alpha = .05):
        _lwr_ = yhat - t.ppf(1 - (alpha / 2), len(self.depv) - 2) * self.residual_stderror(yhati) * \
                    np.sqrt(1. / len(self.indv) + ((x - self.indv.mean()) ** 2) / \
                        (np.sum((self.indv - self.indv.mean()) ** 2) * 1.))
        _upr_ = yhat + t.ppf(1 - (alpha / 2), len(self.depv) - 2) * self.residual_stderror(yhati) * \
                    np.sqrt(1. / len(self.indv) + ((x - self.indv.mean()) ** 2) / \
                        (np.sum((self.indv - self.indv.mean()) ** 2) * 1.))
        return np.column_stack((_lwr_, _upr_))
```

Using this class and its methods, fitting the model to the data is coded as follows:

```
model = linear_regression(x = 'height', y = 'weight', data = women, intercept = True)
print model

# OUTPUT
Model:
        (weight) = -87.517 +  3.450 * (height) + error

Summary:
                        Estimates
        (Intercept)       -87.517
        height              3.450
```

The predicted values of the data points is obtain using the `predict` method,

```
pred = model.predict()
print pred

# OUTPUT
     Predicted       Lower       Upper
0   112.583333  110.963734  114.202933
1   116.033333  114.577601  117.489066
2   119.483333  118.182280  120.784387
3   122.933333  121.774086  124.092581
4   126.383333  125.347718  127.418949
5   129.833333  128.895957  130.770710
6   133.283333  132.410190  134.156477
7   136.733333  135.882678  137.583989
8   140.183333  139.310190  141.056477
9   143.633333  142.695957  144.570710
10  147.083333  146.047718  148.118949
11  150.533333  149.374086  151.692581
12  153.983333  152.682280  155.284387
13  157.433333  155.977601  158.889066
14  160.883333  159.263734  162.502933
```

And Figure 2 below shows the plot of the predicted values along with its confidence interval and data points.

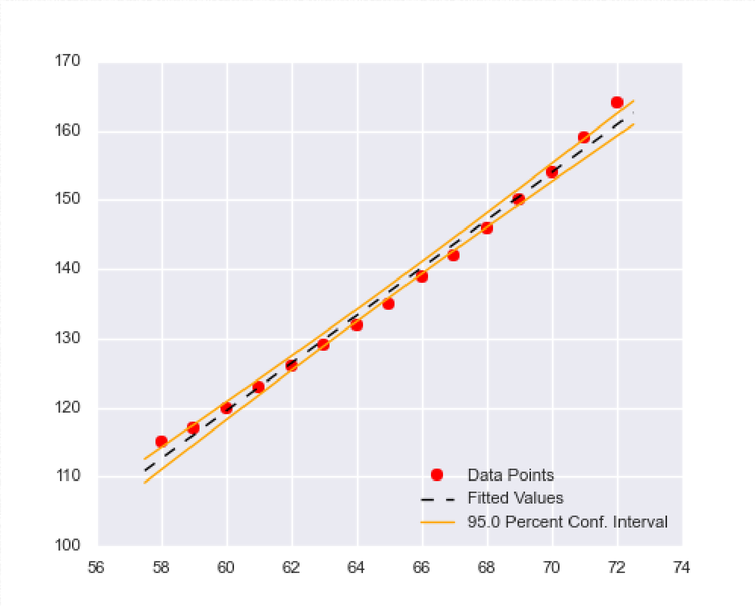


Fig 2. Plot of the Data and the Predicted Values in Python.

```
x = np.linspace(57.5, 72.5)
pred = model.predict(x, plot = True, save_fig = True, filename = 'plot1', fig_format = 'png')
```

If one is only interested on the estimates of the model, then `LinearRegression` of scikit-learn is sufficient, but if the interest on other statistics such as that returned in R model summary is necessary, the said module can also do the job but might need to program other necessary routine. `statsmodels` , on the other hand, returns complete summary of the fitted model as compared to the R output above, which is useful for studies with particular interest on these information. So that modelling the data using simple linear regression is done as follows:

```
import statsmodels.api as sm

X = sm.add_constant(height)
sm_model = sm.OLS(weight, X)
results = sm_model.fit()
print results.summary()

# OUTPUT
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.991
Model:                            OLS   Adj. R-squared:                  0.990
Method:                 Least Squares   F-statistic:                     1433.
Date:                Sun, 09 Aug 2015   Prob (F-statistic):           1.09e-14
Time:                        21:40:25   Log-Likelihood:                -26.541
No. Observations:                  15   AIC:                             57.08
Df Residuals:                      13   BIC:                             58.50
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
const        -87.5167      5.937    -14.741      0.000      -100.343   -74.691
x1             3.4500      0.091     37.855      0.000         3.253     3.647
==============================================================================
Omnibus:                        2.396   Durbin-Watson:                   0.315
Prob(Omnibus):                  0.302   Jarque-Bera (JB):                1.660
Skew:                           0.789   Prob(JB):                        0.436
Kurtosis:                       2.596   Cond. No.                         982.
==============================================================================

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

Clearly, we could spare time with statsmodels, especially in diagnostic checking involving test statistics such as Durbin-Watson and Jarque-Bera tests. We can of course add some plotting for diagnostic, but I prefer to discuss that on a separate entry.

Simple Linear Regression in SAS

Now let's consider running the data in SAS, I am using SAS Studio and in order to import the data, I saved it as a CSV file first with columns height and weight.

Uploaded it to SAS Studio, in which follows are the codes below to import the data.

```
* Import the data;
FILENAME WOMEN "/folders/myfolders/sasuser.v94/women.csv";

PROC IMPORT DATAFILE = WOMEN
    OUT = WORK.WOMEN
    DBMS = CSV;
    GETNAMES = YES;
RUN;
```

Next we fit the model to the data using the `REG` procedure,

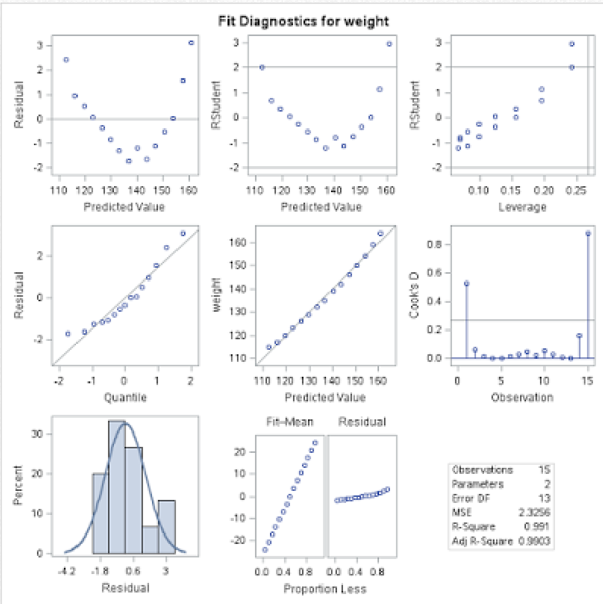
```
PROC REG DATA = WOMEN;
    MODEL weight = height;
RUN;
```

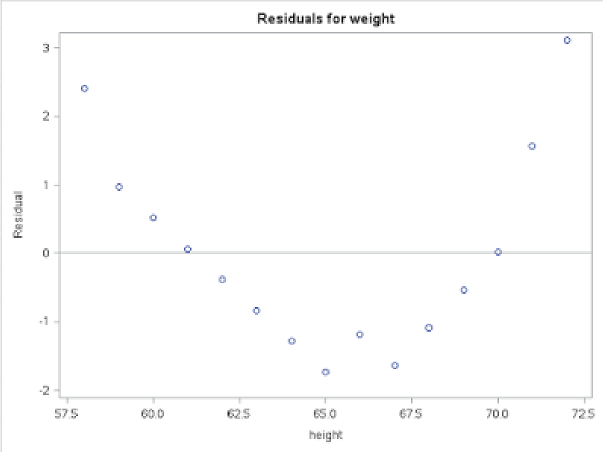
Number of Observations Read	15
Number of Observations Used	15

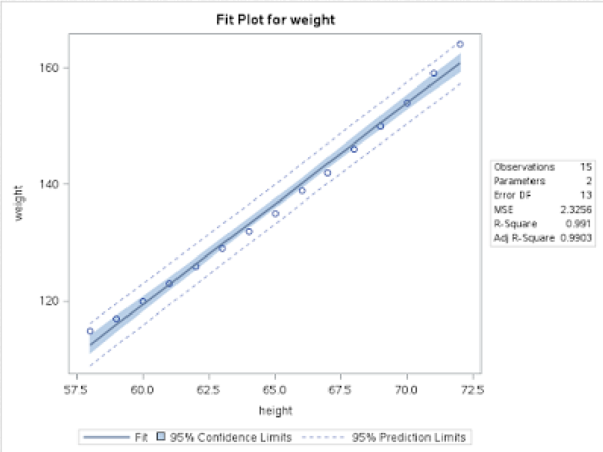
Analysis of Variance					
Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	1	3332.70000	3332.70000	1433.02	<.0001
Error	13	30.23333	2.32564		
Corrected Total	14	3362.93333			

Root MSE	1.52501	R-Square	0.9910
Dependent Mean	136.73333	Adj R-Sq	0.9903
Coeff Var	1.11531		

Parameter Estimates					
Variable	DF	Parameter Estimate	Standard Error	t Value	Pr > |t|
Intercept	1	-87.51667	5.93694	-14.74	<.0001
height	1	3.45000	0.09114	37.86	<.0001







Now that's a lot of output, probably the complete one. But like I said, I am not going to discuss each of these values and plots as some of it are used for diagnostic checking (you can read more on that in reference 1, and in other applied linear regression books). For now, let's just confirm the coefficients obtained -- both the estimates are the same with that in R and Python.

Multiple Linear Regression (MLR)

To extend SLR to MLR, we'll demonstrate this by simulation. Using the formula-based `lm` function of R, assuming we have x_1 and x_2 as our predictors, then following is how we do MLR in R:

```
library(magrittr)

# Simulate the data
x1 <- rnorm(100, 600, 6)
x2 <- rnorm(100, 60, 3)
y <- .35 * x1 + .56 * x2 + rnorm(100)

# Fit the model
mydata <- data.frame(y, x1, x2)
fit <- {y ~ x1 + x2} %>% lm(data = mydata)
fit %>% summary

# OUTPUT
Call:
lm(formula = ., data = mydata)

Residuals:
     Min       1Q    Median       3Q      Max
-2.19496 -0.68206 -0.02526  0.79252  2.73979

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -5.83316    9.98337  -0.584     0.56
x1           0.35989    0.01686  21.343   <2e-16 ***
x2           0.56208    0.03652  15.391   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.082 on 97 degrees of freedom
```

```
Multiple R-squared:  0.8959,    Adjusted R-squared:  0.8937
F-statistic: 417.2 on 2 and 97 DF,  p-value: <
```

Although we did not use intercept in simulating the data, but the obtained estimates for β_1 and β_2 are close to the true parameters (.35 and .56). The intercept, however, will help us capture the noise term we added in simulation.

Next we'll try MLR in Python using statsmodels, consider the following:

```
from numpy import random, column_stack
from statsmodels.api import add_constant, OLS

# Simulate the data
x1 = random.normal(600, 6, 100)
x2 = random.normal(60, 3, 100)
X = column_stack((x1, x2))
X = add_constant(X)
y = .35 * x1 + .56 * x2 + random.normal(size = 100)

# Fit the model
model = OLS(y, X)
fit = model.fit()
print fit.summary()

# OUTPUT
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.868
Model:                            OLS   Adj. R-squared:                  0.866
Method:                 Least Squares   F-statistic:                     319.8
Date:                Thu, 13 Aug 2015   Prob (F-statistic):           1.99e-43
Time:                        17:55:02   Log-Likelihood:                -146.55
No. Observations:                 100   AIC:                             299.1
Df Residuals:                      97   BIC:                             306.9
Df Model:                           2
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [95.0% Conf. Int.]
------------------------------------------------------------------------------
const          7.4352     10.857      0.685      0.495       -14.113    28.984
x1             0.3337      0.018     18.818      0.000         0.298     0.369
x2             0.5975      0.035     17.101      0.000         0.528     0.667
==============================================================================
Omnibus:                        0.923   Durbin-Watson:                   2.178
Prob(Omnibus):                  0.630   Jarque-Bera (JB):                0.916
Skew:                           0.051   Prob(JB):                        0.632
Kurtosis:                       2.542   Cond. No.                     6.15e+04
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
[2] The condition number is large, 6.15e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
```

It should be noted that, the estimates in R and in Python should not (necessarily) be the same since these are simulated values from different software. Finally, we can perform MLR in SAS as follows:

```
* Simulate the data;
PROC IML;
    x1 = J(100, 1);
    x2 = J(100, 1);
    er = J(100, 1);
    CALL RANDGEN(x1, 'NORMAL', 600, 6);
    CALL RANDGEN(x2, 'NORMAL', 60, 3);
    CALL RANDGEN(er, 'NORMAL', 0, 1);

    y = .35 * x1 + .56 * x2 + er;
    df_mat = y || x1 || x2;

    CREATE mydata VAR {y x1 x2};
```

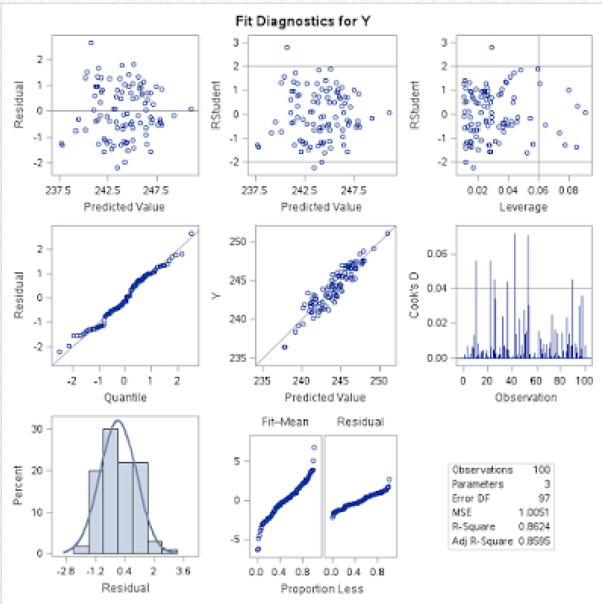
```
    APPEND;
RUN;

* Fit the model;
PROC REG DATA = mydata ;
    MODEL y = x1 x2;
RUN;
```

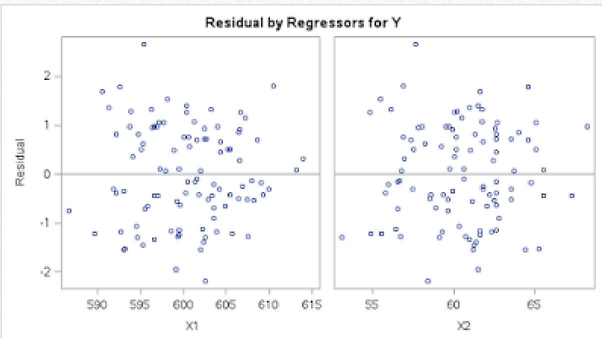
Number of Observations Read	100
Number of Observations Used	100

Analysis of Variance					
Source	DF	Sum of Squares	Mean Square	F Value	Pr > F
Model	2	610.86535	305.43268	303.88	<.0001
Error	97	97.49521	1.00511		
Corrected Total	99	708.36056			

Root MSE	1.00255	R-Square	0.8624
Dependent Mean	244.07327	Adj R-Sq	0.8595
Coeff Var	0.41076		

Parameter Estimates					
Variable	DF	Parameter Estimate	Standard Error	t Value	Pr > |t|
Intercept	1	18.01299	11.10116	1.62	0.1079
X1	1	0.31770	0.01818	17.47	<.0001
X2	1	0.58276	0.03358	17.35	<.0001





Conclusion

In conclusion, R, Python, and SAS estimated the parameters consistently. SAS in particular saves a lot of work, since it returns complete summary of the model, no doubt why companies prefer to use this, besides from their active customer support. R and Python, on the other hand, despite the fact that it is open-source, it can well compete with the former, although it requires programming skills to achieved all of the SAS outputs, but I think that's the exciting part of it -- it makes you think, and manage time. The achievement in R and Python is of course fulfilling. Hope you've learned something, feel free to share your thoughts on the comment below.

Reference

1. Draper, N. R. and Smith, H. (1966). Applied Regression Analysis. John Wiley & Sons, Inc. United States of America.

2. Scikit-learn Documentation
3. Statsmodels Documentation
4. SAS Documentation
5. Delwiche, Lora D., and Susan J. Slaughter. 2012. The Little SAS® Book: A Primer, Fifth Edition. Cary, NC: SAS Institute Inc.
6. Regression with SAS. Institute for Digital Research and Education. UCLA. Retrieved August 13, 2015.
7. Python Plotly Documentation