Regression Analysis with One Predictor

David Gerbing
The School of Business
Portland State University
gerbing@pdx.edu

Table of Contents

This template shows how to do regression analysis with a single predictor using the `statsmodel` package. The `statsmodel` functions do general statistical analysis, including regression. This week we move beyond just Python to focus on understanding the concept of regression analysis as our introduciton to machine learning. Next week we do a regression analysis with the more useful multiple regression models, multiple predictors, using the most popular Python machine learning framework.

Preliminaries

```
from datetime import datetime as dt
now = dt.now()
print ("Analysis on", now.strftime("%Y-%m-%d"), "at", now.strftime("%H:%M"))
```

```
Analysis on 2021-07-05 at 00:45
```

```
import os
os.getcwd()
```

```
'/content'
```



```
import matplotlib.pyplot as plt
import seaborn as sns
```

Read the Data

The data consist of variables based on regions of Boston from some decades back, with a focus on houses and housing prices.

```
#d = pd.read_csv('data/Boston.csv')
d = pd.read_csv('http://web.pdx.edu/~gerbing/data/Boston.csv')
```

```
d.shape
```

```
(506, 15)
```

```
d.head()
```

	Unnamed: 0	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio
0	1	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3
1	2	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8
2	3	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8
3	4	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7

Do not need the first column, so drop.

```
d = d.drop(["Unnamed: 0"], axis="columns")
d.head()
```

	crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	black	lst
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.

Do a missing data check before analysis.

```
d.isnull().sum()
```

```
crim       0
zn         0
indus      0
chas       0
nox        0
rm         0
age        0
dis        0
rad        0
tax        0
ptratio    0
black      0
lstat      0
medv       0
dtype: int64
```

No missing data here, so can proceed as is.

Form X and y Data Structures

Build a model that forecasts/explains the median house price, *medv* in terms of the average number of rooms, *rm*.

- *medv*: Median value of owner-occupied homes in $1000's
- *rm*: Average number of rooms per dwelling

Store the features, the predictor variables, of which there is only one in this example, in data structure X. Store the target variable in data structure y. The uppercase X is used because in real-world applications, X invariably contains multiple variables.

```
y = d['medv']
X = d['rm']
```

A technical point, but one worth considering when doing data analysis, is to understand the type of data structures created throughout an analysis. The data as read are read into a `pandas` data structure called a *dataframe*. However, when the data frame is sub-setted into X and y, both of which consist of only a single variable in this example, the result is a one-dimensional `pandas` data structure called a `series`. Actually, a *dataframe* consists of columns, and each column is a `series`. That is why reduction of the data frame to a single column results in a `series`.

For this particular analysis pursued here, being aware of this distinction is not necessary. But, in general, always good to know the underlying data structures. Thinking the data is of one type, when it is actually of another type, in many situations leads to programming errors.

To check the type of a variable, use the Python function `type()`.

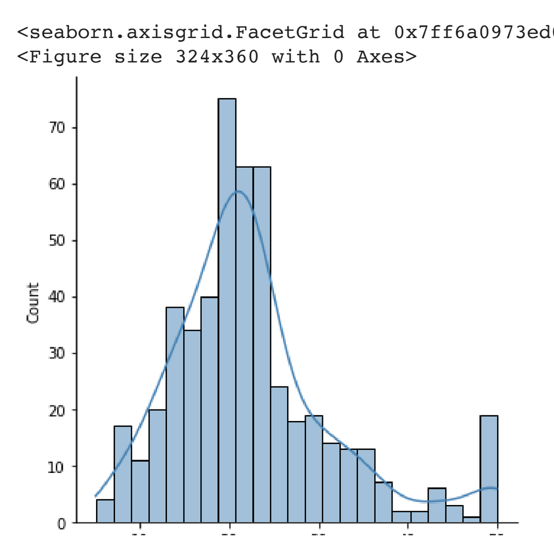
```
print("d: ", type(d))
print("X: ", type(X))
print("y: ", type(y))
```

```
d:  <class 'pandas.core.frame.DataFrame'>
X:  <class 'pandas.core.series.Series'>
y:  <class 'pandas.core.series.Series'>
```

Understand the distribution of the target variable, *medv*, to make sure that the distribution is not too weird. Show the distribution with its histogram and density estimate (smoothed histogram), obtained with the `seaborn` method `displot()`. Get the smoothed summary curve (called a density function) by setting the `kde` parameter to `True`.

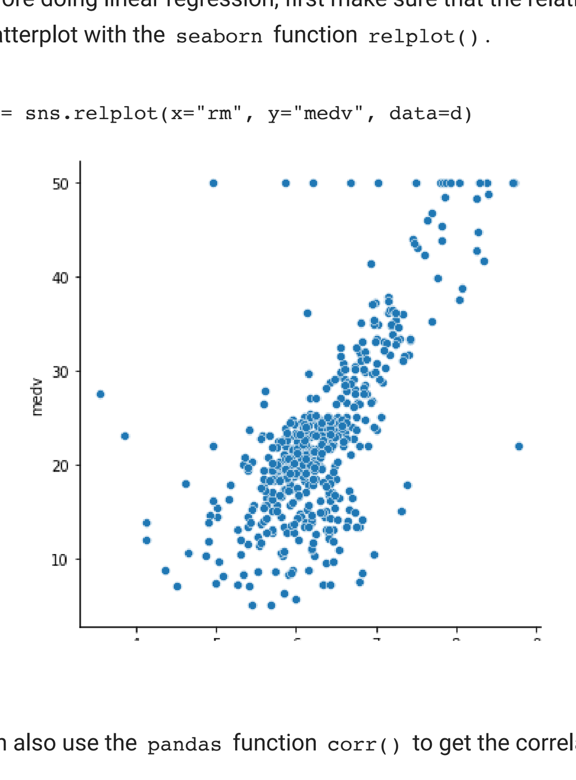
```
plt.figure(figsize=(4.5,5))
sns.displot(x='medv', color='steelblue', kde=True, data=d)
```

```
<seaborn.axisgrid.FacetGrid at 0x7ff6a0973ed0>
<Figure size 324x360 with 0 Axes>
```



Before doing linear regression, first make sure that the relationship between the x and y variables is at least roughly linear. Check via a scatterplot with the `seaborn` function `relplot()`.

```
ax = sns.relplot(x="rm", y="medv", data=d)
```



Can also use the `pandas` function `corr()` to get the correlation between predictor and target.

```
d['rm'].corr(d['medv']).round(2)
```

```
0.7
```

The variables are highly correlated with $r = 0.70$, and the scatterplot indicates an apparent linear relationship. The only "weird" issue is that apparently housing prices over 50,000 USD are truncated and listed at 50,000 USD. Probably a good idea to filter these rows of data out of the data table and generalize the results to houses with less than that value, but we will leave for now.

Model Analysis

Estimation

For some reason, by default, the estimation procedure assumes a y-intercept of 0 unless there is constant value in the feature data. To compensate, before estimating the model, explicitly add a column of 1.0's to the X data structure so that the estimated model will have a y-intercept, and therefore fit better without requiring the assumption of a value of 0. Add the constant with the `statsmodels` package `add_constant()` function.

```
import statsmodels.api as sm
from statsmodels.regression.linear_model import RegressionResults
X = sm.add_constant(X)
X.head()
```

```
/usr/local/lib/python3.7/dist-packages/statsmodels/tools/_testing.py:19: FutureWa
  import pandas.util.testing as tm
```

	const	rm
0	1.0	6.575
1	1.0	6.421
2	1.0	7.185
3	1.0	6.998
4	1.0	7.147

Specify the model with the `OLS()` function from the `statsmodel` package. OLS means *ordinary least squares*, the default and usual estimation procedure for regression models. Specify the target variable first, followed by the data structure with the features. Do the least-squares regression analysis of the defined model by applying the `fit()` function.

Save the results of this regression analysis to a `statsmodel` data structure we name *results*. The `summary()` function summarizes the main results of the analysis.

```
model = sm.OLS(y, X)
results = model.fit()
print(results.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                   medv   R-squared:                       0.484
Model:                            OLS   Adj. R-squared:                  0.483
Method:                 Least Squares   F-statistic:                     471.8
Date:                Mon, 05 Jul 2021   Prob (F-statistic):           2.49e-74
Time:                        00:45:56   Log-Likelihood:                -1673.1
No. Observations:                 506   AIC:                             3350.
Df Residuals:                     504   BIC:                             3359.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        -34.6706      2.650    -13.084      0.000     -39.877     -29.465
rm             9.1021      0.419     21.722      0.000       8.279       9.925
==============================================================================
Omnibus:                      102.585   Durbin-Watson:                   0.684
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              612.449
Skew:                           0.726   Prob(JB):                    1.02e-133
Kurtosis:                       8.190   Cond. No.                         58.4
==============================================================================

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

The primary results of the model are in this section:

```
               coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const      -34.6706      2.650    -13.084      0.000     -39.877     -29.465
rm           9.1021      0.419     21.722      0.000       8.279       9.925
```

The estimated values of the intercept, b_0 and b_1 are -34.67 and 9.10, respectively. So write the estimated model as:

$$\hat{y}_{medv} = -34.67 + 9.10(x_{rm})$$

In *this data set* only, for each increase in the average number of rooms, the average selling price increases by 9,102 USD. We do, however, need inferential statistics to generalize to the population as a whole.

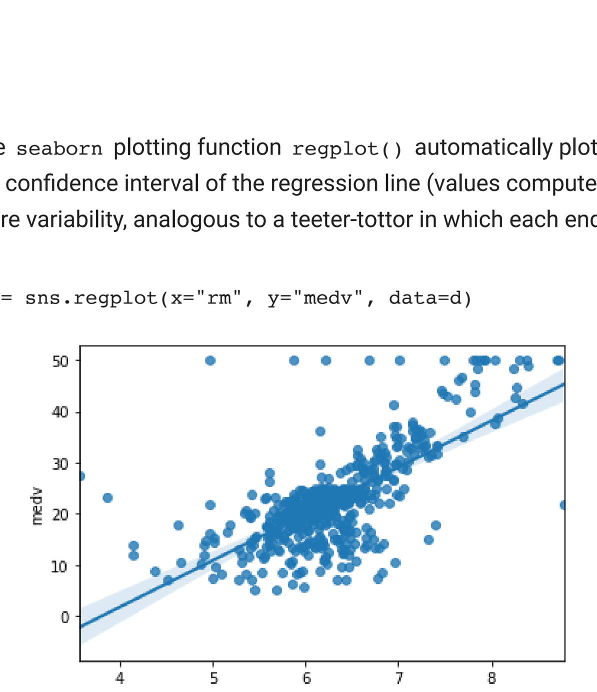
Hypothesis test: The p-values for the t-statistics for each of the two estimated coefficients, `P>|t|`, are well below the cutoff of $\alpha = 0.05$. For the slope, assuming that there is no relationship between *rm* and *medv*, then the probability of getting an estimated slope coefficient as large as 9.10 is an extremely improbable event, indistinguishable from 0 to three decimal digits. So reject the null hypothesis of no relationship, and conclude there is a positive relationship between *rm* and *medv*. As *rm* increases, so does *medv*.

Confidence interval: Accordingly, the 95% confidence interval for the slope, which contains the plausible values of the true, population value of the slope coefficient, is, with 95% confidence, in the interval from 8.279 to 9.925. That is, with 95% confidence, for each increase in the average number of rooms, the average selling price increases somewhere between 8,279 USD and 9,925 USD.

Fit

The `seaborn` plotting function `regplot()` automatically plots the scatterplot and the regression line through the scatterplot. Also provided is the confidence interval of the regression line (values computed over hypothetical repeated samples). Values of X futher from the middle have more variability, analogous to a teeter-tottor in which each end swings further than a place on the teeter-tottor closer to the middle, the fulcrum.

```
ax = sns.regplot(x="rm", y="medv", data=d)
```



From the line through the scatterplot, fit looks reasonable. There is, of course, scatter about the line, but not so much.

To evaluate fit of the model with statistics, access various values computed by the `fit()` function, here stored in the structure called *results*. Use the functions `ssr()`, `mse_resid()`, and `rsquared()`. The RMSE or standard deviation of the residuals is not provided directly, so compute as the square root of the MSE. Use the function `sqrt()` from the `numpy` package. All values are rounded to two decimal digits with the `round()` function.

```
print("Sum of squared residuals:", results.ssr.round(2))
print("Mean squared error:", results.mse_resid.round(2))
RMSE = np.sqrt(results.mse_resid)
print("Stdev of residuals:", RMSE.round(2))
res_range = 4 * RMSE
print("95% range of residuals:", res_range.round(2))
print("R-squared:", results.rsquared.round(2))
```

```
Sum of squared residuals: 22061.88
Mean squared error: 43.77
Stdev of residuals: 6.62
95% range of residuals: 26.46
R-squared: 0.48
```

The sum of the squared residuals is provided for reference, upon which the more meaningful fit indices are derived. The standard deviation of the residuals, consistent with not too much scatter about the scatterplot, is fairly small, indicating reasonable fit, with 95% of residuals spanning a range of about 26 and 1/2 USD.

R^2 is almost 0.5, which indicates room for improvement, but a demonstration that the model improves much over the null model. That is, using *rm* to predict *medv* does much better than simply using the mean of *medv* to predict *medv*.

Postscript

In practice, regression models, and all machine learning models, typically involve much more than a single feature, predictor variable. Subsequent material expands this model to multiple regression, that is, multiple features.