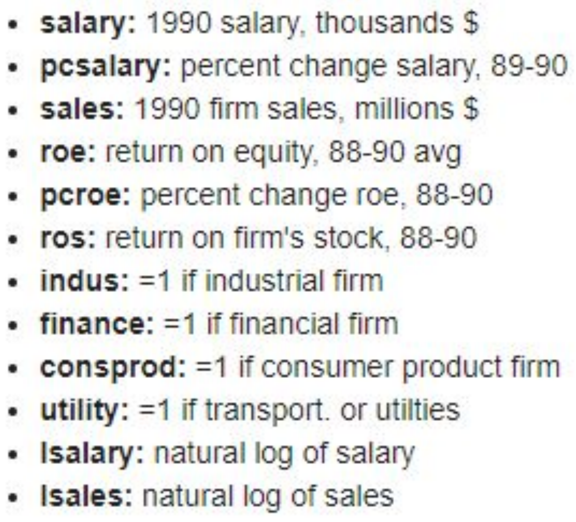
Regression (Continued)

Today’s Outline

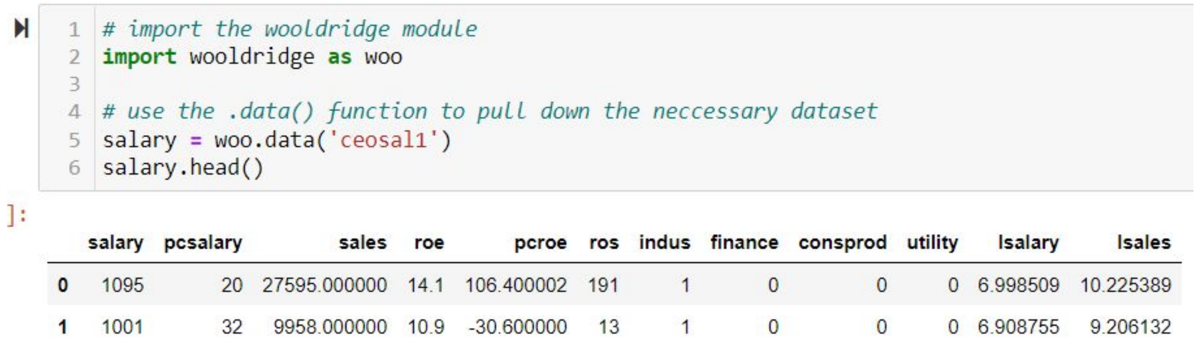
- Conditional Distribution
- Simple Regression
  - Simple linear regression assumptions
  - Strict exogeneity
  - Heteroskedasticity
  - Error Normality
  - Standard Errors
  - Prediction vs Confidence intervals

Example Dataset

- Dataset used to investigate the relationship between CEO salary and sales
- Sample of data was reported in the May 6, 1991 issue of Businessweek



- **salary:** 1990 salary, thousands $
- **pcsalary:** percent change salary, 89-90
- **sales:** 1990 firm sales, millions $
- **roe:** return on equity, 88-90 avg
- **pcroe:** percent change roe, 88-90
- **ros:** return on firm's stock, 88-90
- **indus:** =1 if industrial firm
- **finance:** =1 if financial firm
- **consprod:** =1 if consumer product firm
- **utility:** =1 if transport. or utilties
- **lsalary:** natural log of salary
- **lsales:** natural log of sales

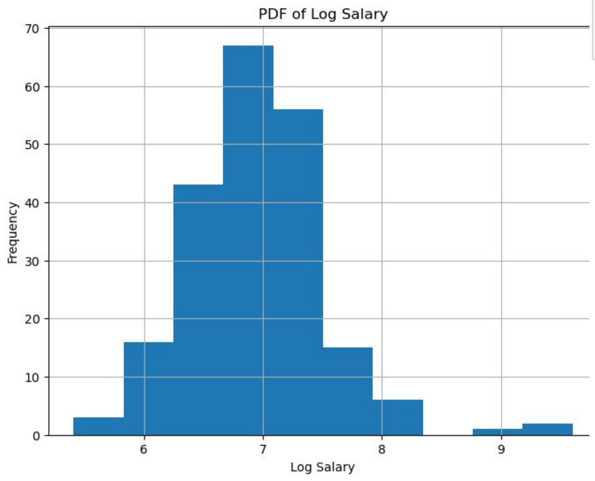


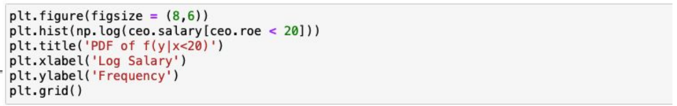
```python
# import the wooldridge module
import wooldridge as woo

# use the .data() function to pull down the neccessary dataset
salary = woo.data('ceosal1')
salary.head()
```

	salary	pcsalary	sales	roe	pcroe	ros	indus	finance	consprod	utility	lsalary	lsales
0	1095	20	27595.000000	14.1	106.400002	191	1	0	0	0	6.998509	10.225389
1	1001	32	9958.000000	10.9	-30.600000	13	1	0	0	0	6.908755	9.206132

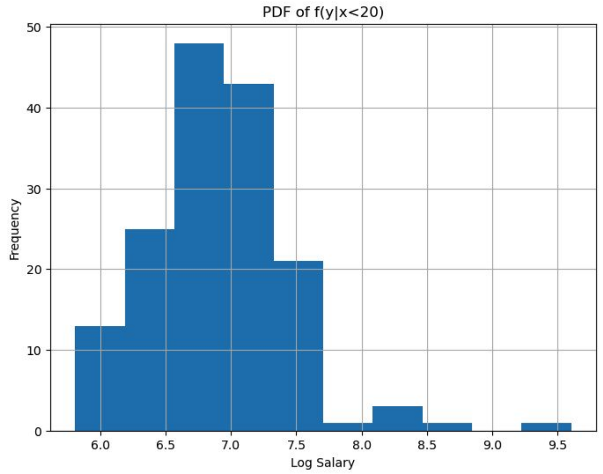
Conditional Distributions

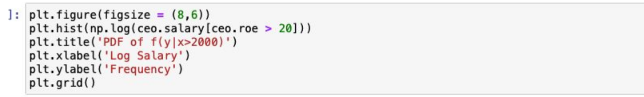
- A simple linear regression is based on assumptions about the conditional pdf of some dependent variable y (Salary) on an independent variable (x)
- We can use boolean indexing to get the pdf of salary from our data conditional on some value of return on equity



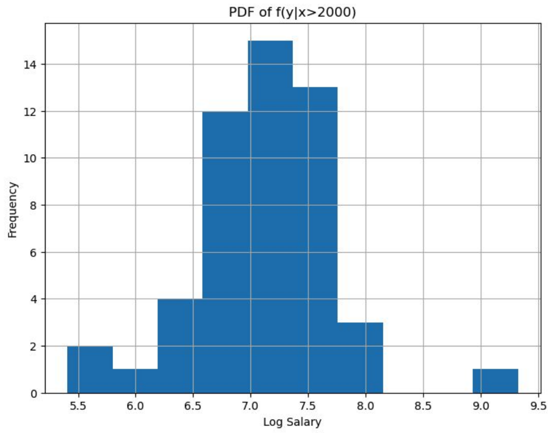


```
plt.figure(figsize = (8,6))
plt.hist(np.log(ceo.salary[ceo.roe < 20]))
plt.title('PDF of f(y|x<20)')
plt.xlabel('Log Salary')
plt.ylabel('Frequency')
plt.grid()
```





```
]: plt.figure(figsize = (8,6))
   plt.hist(np.log(ceo.salary[ceo.roe > 20]))
   plt.title('PDF of f(y|x>2000)')
   plt.xlabel('Log Salary')
   plt.ylabel('Frequency')
   plt.grid()
```



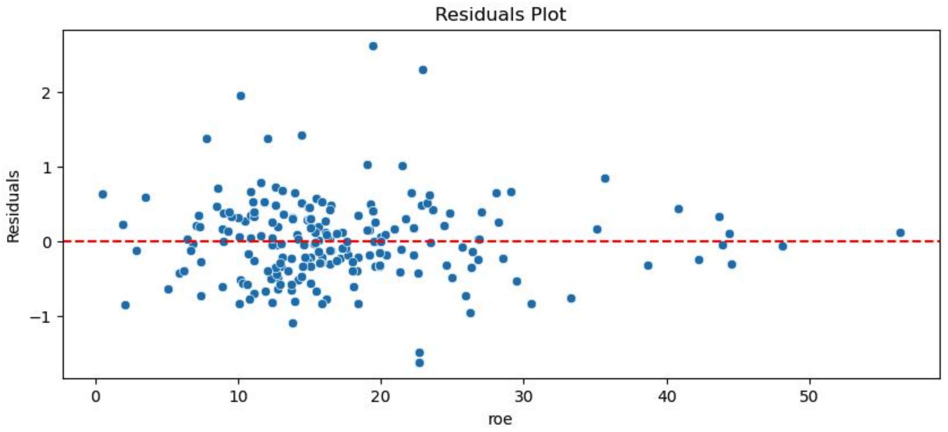
Regression and Residuals Plot

- We can also look at the estimates of our residuals (distance of our observations from the regression line)
- For a simple regression we will plot our dependent variable on the x-axis against the residuals on the y-axis
- This can be generated using any scatter plot, this will be useful for checking important properties of our regression

```
# fit a regression - note that we can add a log transform
# using numpy in the equation
reg = smf.ols('np.log(salary) ~ roe', data = ceo)
results = reg.fit()
results.params
```

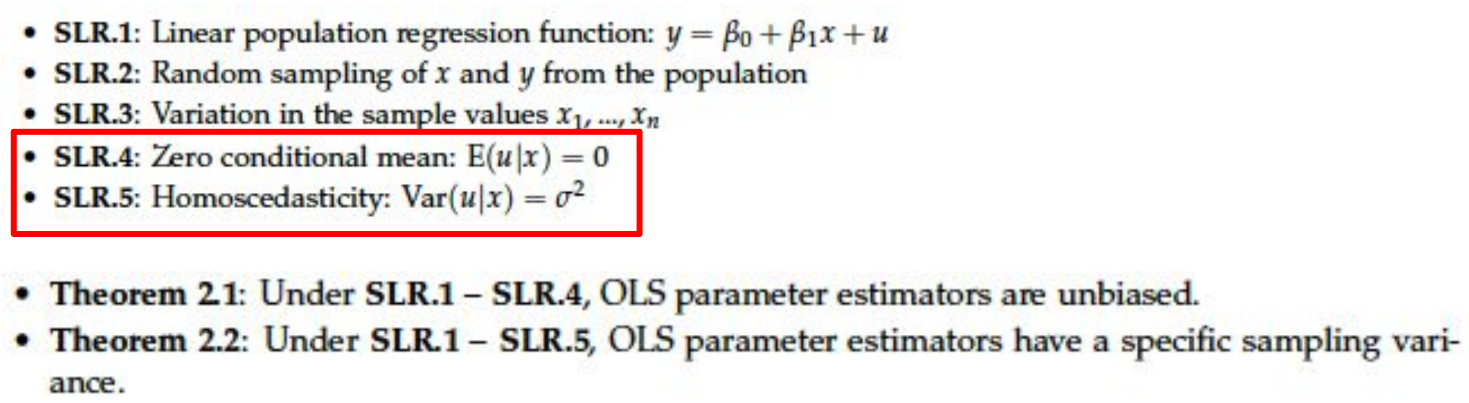
```
Intercept    6.712169
roe          0.013863
dtype: float64
```

```
# fit regression to synthetic data
plt.figure(figsize = (10, 4))

# we can get the original data as an attribute as well as the residuals
sns.scatterplot(x = ceo.roe, y = results.resid)
plt.axhline(0, linestyle = '--', color = "red")
plt.ylabel("Residuals")
plt.title("Residuals Plot")
plt.show()
```



Simple Linear Regression Assumptions

- There are five classic assumptions for simple linear regression that will ensure our estimates of the slope and intercept are unbiased and the minimal possible variance
- The most important assumptions to pay attention to will be SLR 4 and 5



- **SLR.1**: Linear population regression function: $y = \beta_0 + \beta_1 x + u$
- **SLR.2**: Random sampling of x and y from the population
- **SLR.3**: Variation in the sample values $x_1, \ldots, x_n$
- **SLR.4**: Zero conditional mean: $\mathrm{E}(u|x) = 0$
- **SLR.5**: Homoscedasticity: $\mathrm{Var}(u|x) = \sigma^2$

- **Theorem 2.1**: Under **SLR.1 – SLR.4**, OLS parameter estimators are unbiased.
- **Theorem 2.2**: Under **SLR.1 – SLR.5**, OLS parameter estimators have a specific sampling variance.

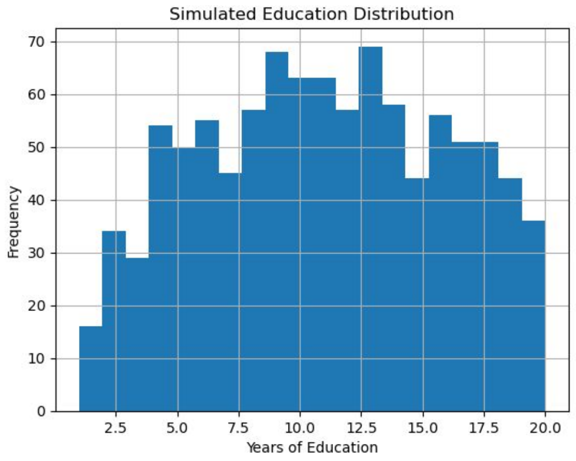
Strict Exogeneity

- SLR states that the value of the residuals should not be dependent on the value of your predictor
- If this assumption does not hold then your parameter estimates will be biased (interview question)

$$E[e_i | x_i] = 0$$

- For example, suppose the true relationship between income and education was:

$$\text{Income (Thousands)} = 40 + 5 * \text{Education (years)} + u$$



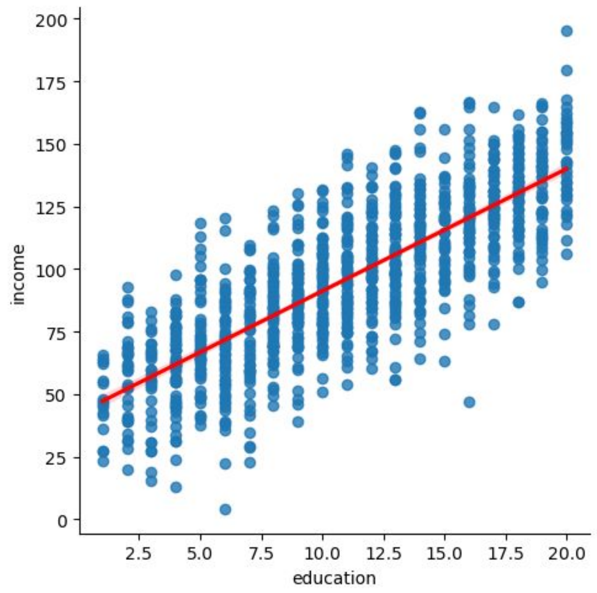
Simulated Exogeneity

- We can run the regression in the case where the errors (u) do not depend on education (x) to get an unbiased estimate of the population parameters

```
# add the true popultaion parameters
beta0 = 40
beta1 = 5
su = 20

# create some noise and a dependent variable
u = np.random.normal(0, su, n)
education = nums

# create a synthetic y-variable
income = beta0 + beta1*education + u
```

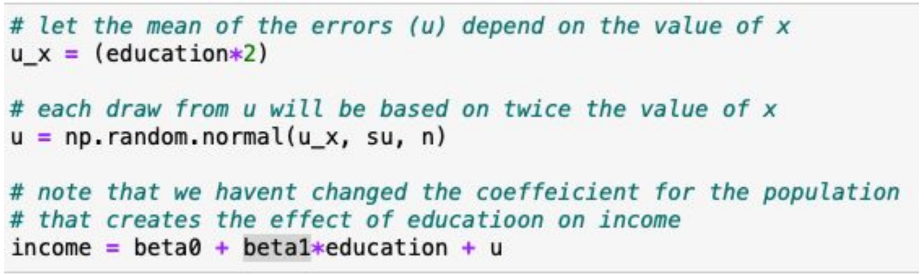
```
results = smf.ols('income ~ education', data = data).fit()

results.params
```

```
Intercept    42.352327
education     4.884671
dtype: float64
```

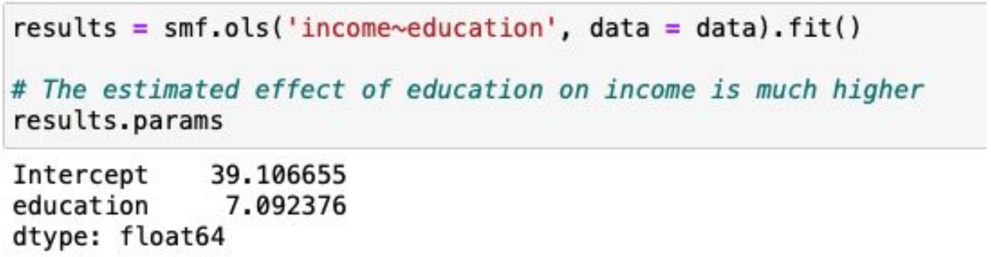


Violation of SLR 4

- However in the real world, there are likely factors in u that are correlated with x
  - Can you think of some examples? (factors that may effect education *and* income that may obscure the true estimate)
- If the value of u does depend somewhat on x, then our estimates will be biased
- What direction do you think the estimate of the effect of education on income would likely be biased?

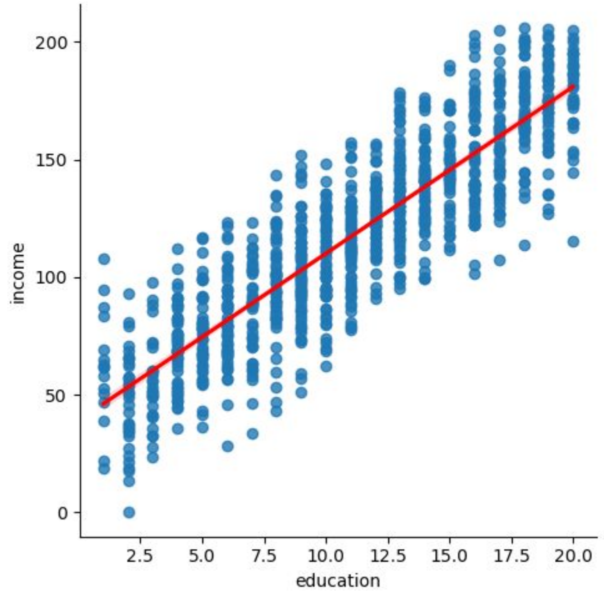


```
# let the mean of the errors (u) depend on the value of x
u_x = (education*2)

# each draw from u will be based on twice the value of x
u = np.random.normal(u_x, su, n)

# note that we havent changed the coeffeicient for the population
# that creates the effect of educatioon on income
income = beta0 + beta1*education + u
```



```
results = smf.ols('income~education', data = data).fit()

# The estimated effect of education on income is much higher
results.params
```

```
Intercept    39.106655
education     7.092376
dtype: float64
```

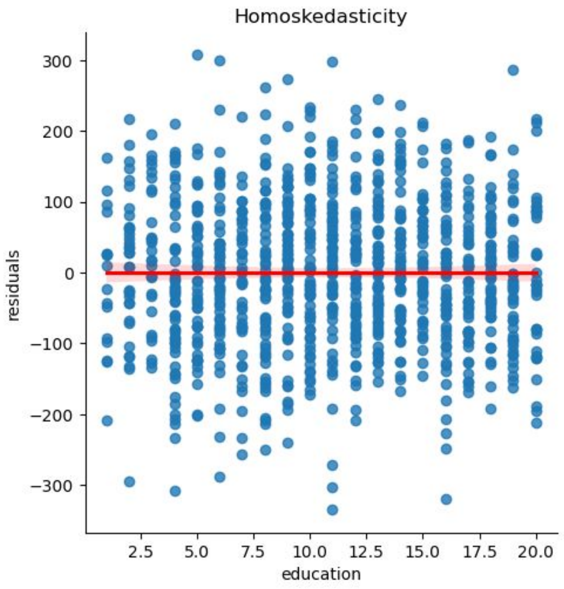


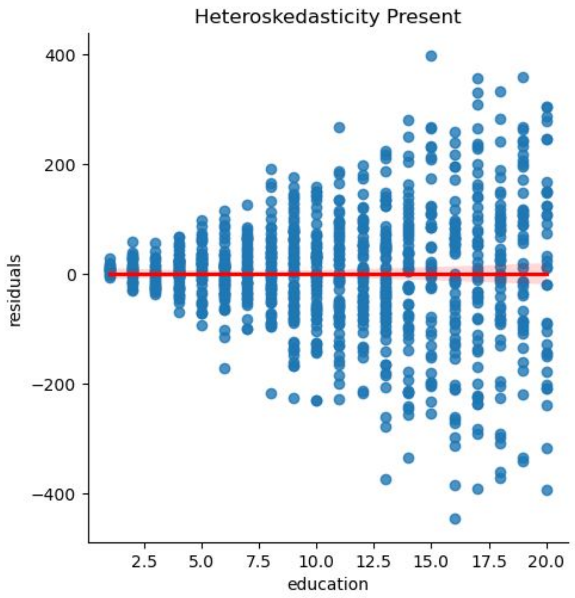
Heteroscedasticity (SLR 5)

- SLR 5 is the assumption that the variance of the residuals does not depend on x

- **SLR.5**: Homoscedasticity: $\text{Var}(u|x) = \sigma^2$

- Unlike for usual violations of SLR 4, we can often identify the presence of heteroskedasticity by looking at an X vs residuals plot for simple linear regression



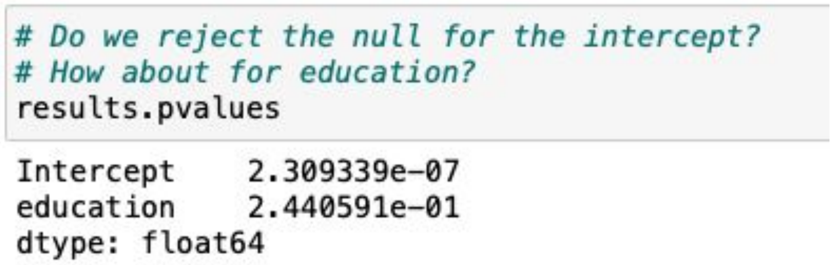


Statistical Significance and P-Values in Python

- The consequence of heteroskedasticity on our regression will be present in our variance
- Variance is used for deciding whether the effect of our dependent variable on our independent variable is *statistically significant*
- We are testing the null hypothesis that:

$$H_0 : \beta_k = 0 \text{ vs } H_1 : \beta_k \neq 0$$

- In python we do this by looking at the p-values of the regression coefficient and commonly say a coefficient is statistically significant if it is less than .05 (lower is more restrictive

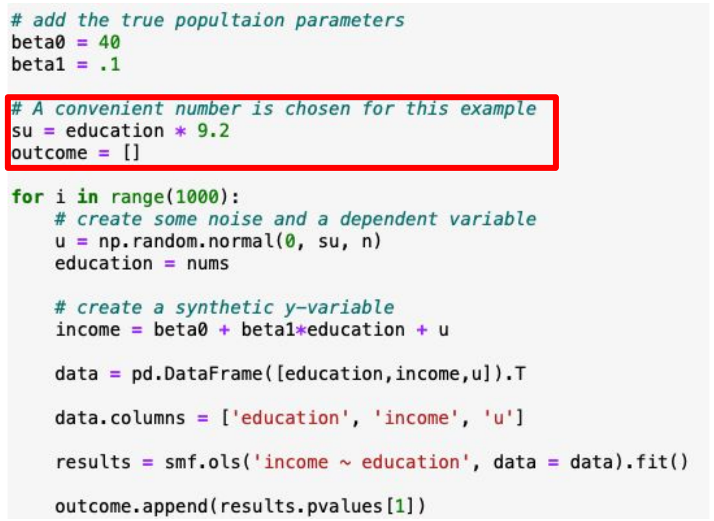


```
# Do we reject the null for the intercept?
# How about for education?
results.pvalues
```

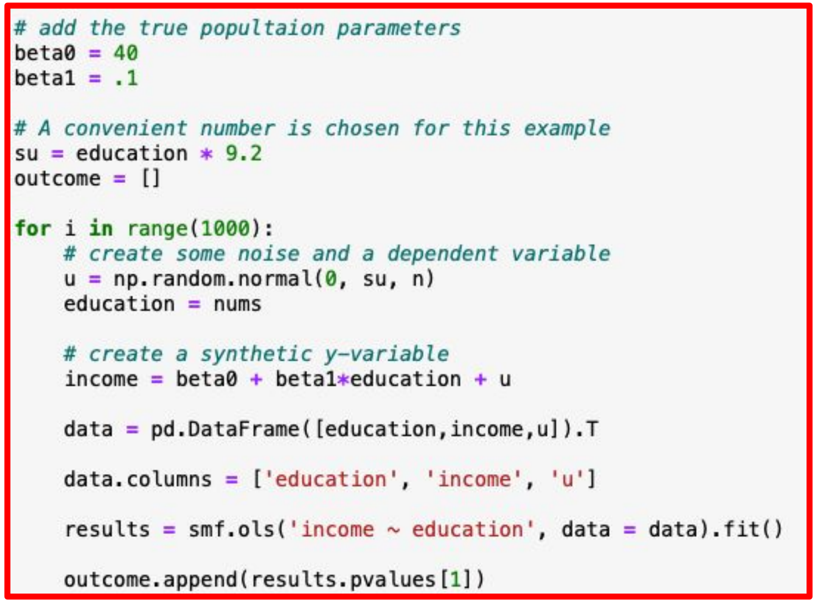
```
Intercept    2.309339e-07
education    2.440591e-01
dtype: float64
```

Statistical Significance and Violations of Homoskedasticity

- Since heteroskedasticity will impact our variance, and p-values are a function of variance, we can no longer trust them
- We also cannot trust other measures based on variance like confidence intervals
- Heteroskedasticity may cause us to over or under reject our null hypothesis
- Below we simulate 1000 hypothesis test for the case where the variance of the errors depends on x (left) and when it does not(right)



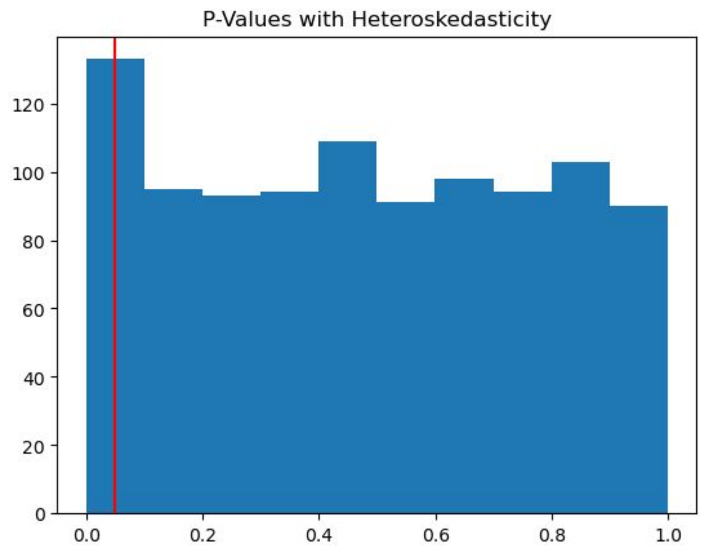
```python
# add the true popultaion parameters
beta0 = 40
beta1 = .1

# A convenient number is chosen for this example
su = education * 9.2
outcome = []

for i in range(1000):
    # create some noise and a dependent variable
    u = np.random.normal(0, su, n)
    education = nums

    # create a synthetic y-variable
    income = beta0 + beta1*education + u

    data = pd.DataFrame([education,income,u]).T

    data.columns = ['education', 'income', 'u']

    results = smf.ols('income ~ education', data = data).fit()

    outcome.append(results.pvalues[1])
```

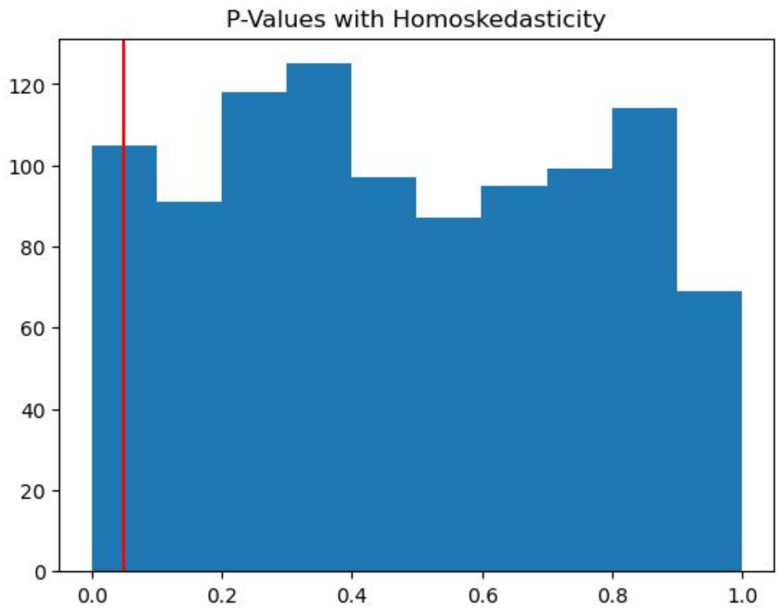


```python
# add the true popultaion parameters
beta0 = 40
beta1 = .1

# A convenient number is chosen for this example
su = education * 9.2
outcome = []

for i in range(1000):
    # create some noise and a dependent variable
    u = np.random.normal(0, su, n)
    education = nums

    # create a synthetic y-variable
    income = beta0 + beta1*education + u

    data = pd.DataFrame([education,income,u]).T

    data.columns = ['education', 'income', 'u']

    results = smf.ols('income ~ education', data = data).fit()

    outcome.append(results.pvalues[1])
```

Statistical Significance and Violations of Homoskedasticity

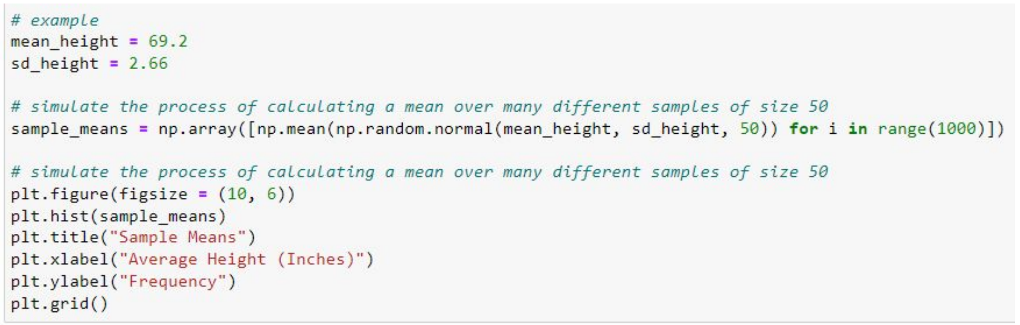
- Below we plot histograms of the p-values from each experiment
- Note that we are much more likely (nearly 2x) to reject the null hypothesis when heteroskedasticity is present than when it is not



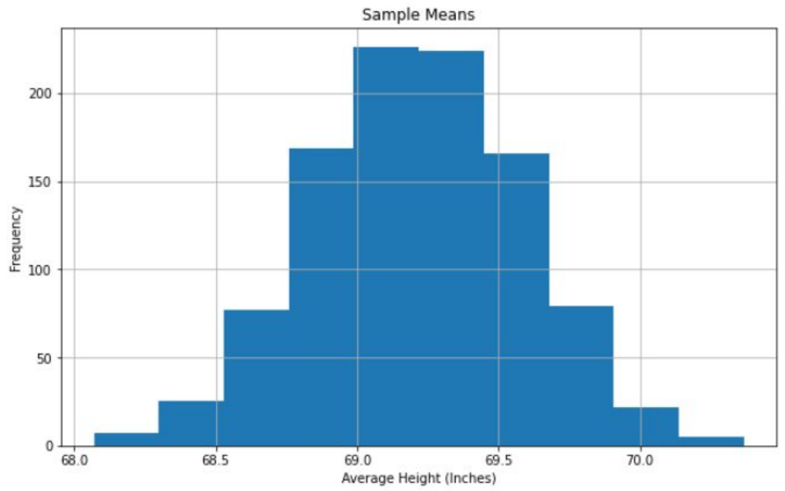


Simulating Sampling Variability

- Statistics such as our regression coefficients or sample mean are *random variables*
- Their values are dependent on a random sample
- We would like to understand how much the value of these statistics could vary between samples of the same size
- On the right I draw repeated samples from the theoretical distribution of male heights and find the mean



```
# example
mean_height = 69.2
sd_height = 2.66

# simulate the process of calculating a mean over many different samples of size 50
sample_means = np.array([np.mean(np.random.normal(mean_height, sd_height, 50)) for i in range(1000)])

# simulate the process of calculating a mean over many different samples of size 50
plt.figure(figsize = (10, 6))
plt.hist(sample_means)
plt.title("Sample Means")
plt.xlabel("Average Height (Inches)")
plt.ylabel("Frequency")
plt.grid()
```



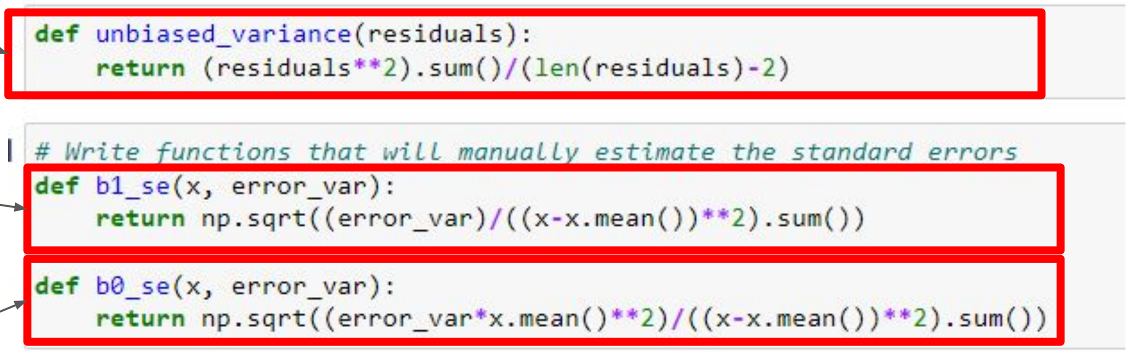
Error Variance and Standard Errors

- Below we generate functions designed to estimate the:
  - Error variance (how much variation there is in the unobservables affecting y)
  - Standard error of beta1
  - Standard error of beta2

$$\hat{\sigma}^2 = \frac{1}{n-2}\sum_{i=1}^{n}\hat{u}_i^2$$

$$se\left(\hat{\beta}_1\right) = \sqrt{\frac{\hat{\sigma}^2}{\sum_{i=1}^{n}(x-\bar{x})^2}}$$

$$se\left(\hat{\beta}_0\right) = \sqrt{\frac{\hat{\sigma}^2\bar{x}^2}{\sum_{i=1}^{n}(x-\bar{x})^2}}$$

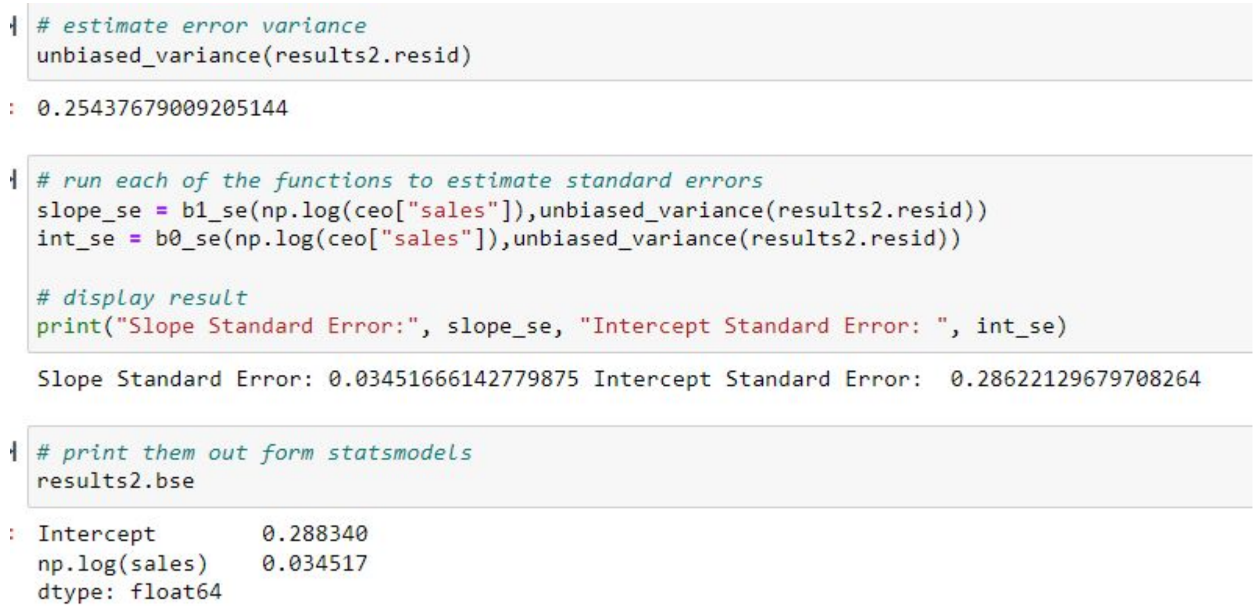


```python
def unbiased_variance(residuals):
    return (residuals**2).sum()/(len(residuals)-2)
```

```python
# Write functions that will manually estimate the standard errors
def b1_se(x, error_var):
    return np.sqrt((error_var)/((x-x.mean())**2).sum())

def b0_se(x, error_var):
    return np.sqrt((error_var*x.mean()**2)/((x-x.mean())**2).sum())
```

Error Variance and Standard Errors (Calculation)

- Below I print the error variance and standard errors of each coefficient for a log-log regression of ceo salary on sales
- Note the standard errors and error variance can be pulled from the regression results



```
# estimate error variance
unbiased_variance(results2.resid)
```

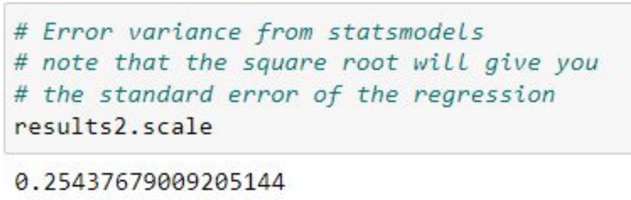
```
0.25437679009205144
```

```
# run each of the functions to estimate standard errors
slope_se = b1_se(np.log(ceo["sales"]),unbiased_variance(results2.resid))
int_se = b0_se(np.log(ceo["sales"]),unbiased_variance(results2.resid))

# display result
print("Slope Standard Error:", slope_se, "Intercept Standard Error: ", int_se)
```

```
Slope Standard Error: 0.03451666142779875 Intercept Standard Error:  0.28622129679708264
```

```
# print them out form statsmodels
results2.bse
```

```
Intercept         0.288340
np.log(sales)     0.034517
dtype: float64
```



```
# Error variance from statsmodels
# note that the square root will give you
# the standard error of the regression
results2.scale
```

```
0.25437679009205144
```

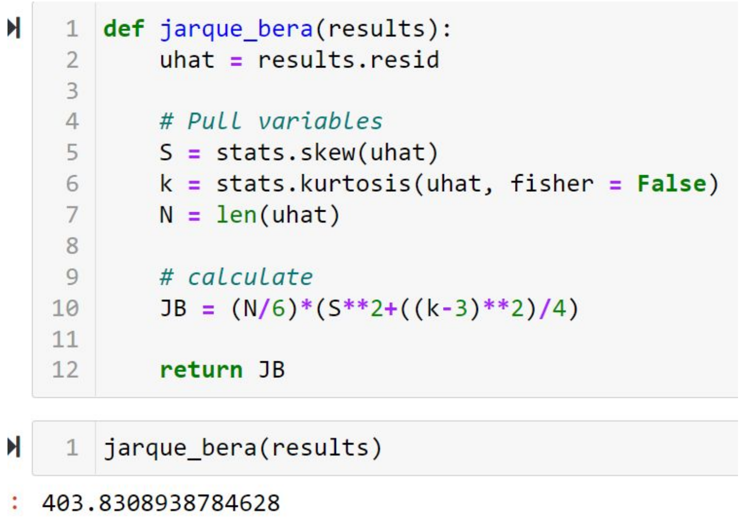
Jarque-Bera Test

- Normality of your residuals is nice to have, especially when the sample isn't large
- Certain transformations can help make residuals more normal
- The Jarque-Bera test can help us determine whether the residuals are likely to be normal
- The null hypothesis is H_0: JB = 0, why?

$$JB = \frac{N}{6}(S^2 + \frac{(k-3)^2}{4})$$

Running Jarque-Bera Test

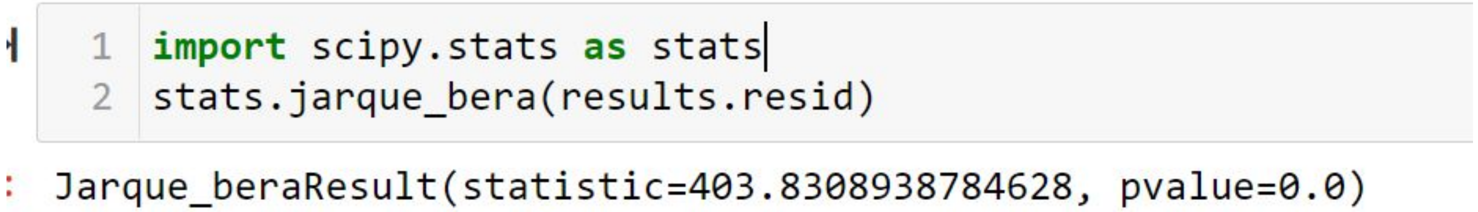
- The formula is simple to implement directly
  - Outputs test statistic
- Scipy.stats also has a function that can implement the test
  - Outputs the test statistic and p-value



```python
def jarque_bera(results):
    uhat = results.resid

    # Pull variables
    S = stats.skew(uhat)
    k = stats.kurtosis(uhat, fisher = False)
    N = len(uhat)

    # calculate
    JB = (N/6)*(S**2+((k-3)**2)/4)

    return JB
```

```python
jarque_bera(results)
```

```
403.8308938784628
```

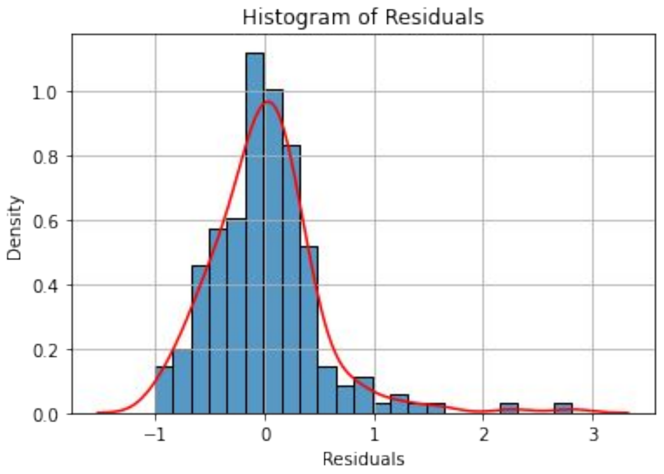


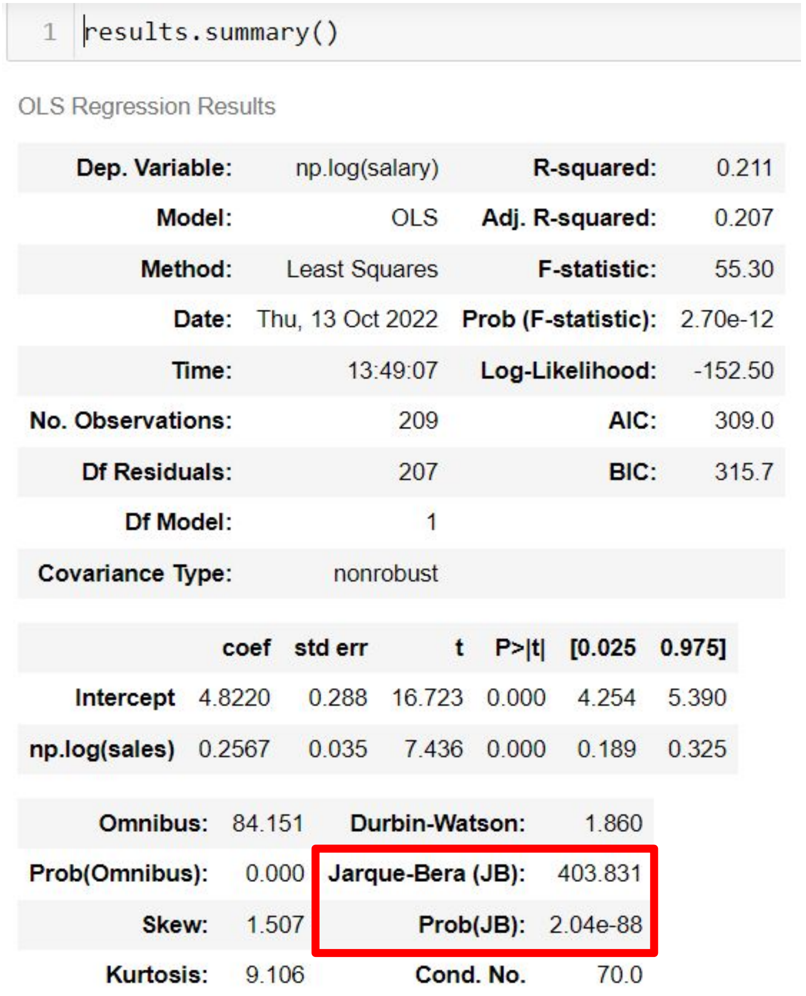
```python
import scipy.stats as stats
stats.jarque_bera(results.resid)
```

```
Jarque_beraResult(statistic=403.8308938784628, pvalue=0.0)
```

Jarque-Bera Test in Summary

- The summary module also outputs several useful statistics about the residuals
  - Skew
  - Kurtosis
  - JB test





```
results.summary()
```

OLS Regression Results

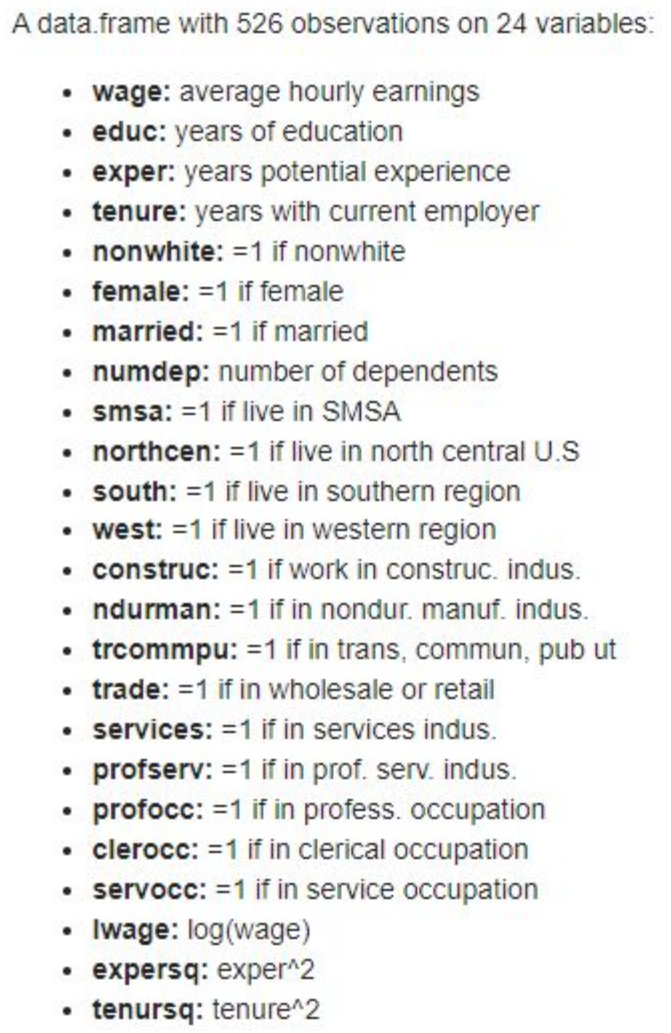
Dep. Variable:	np.log(salary)	**R-squared:**	0.211
Model:	OLS	**Adj. R-squared:**	0.207
Method:	Least Squares	**F-statistic:**	55.30
Date:	Thu, 13 Oct 2022	**Prob (F-statistic):**	2.70e-12
Time:	13:49:07	**Log-Likelihood:**	-152.50
No. Observations:	209	**AIC:**	309.0
Df Residuals:	207	**BIC:**	315.7
Df Model:	1		
Covariance Type:	nonrobust		

	coef	**std err**	**t**	**P>|t|**	**[0.025**	**0.975]**
Intercept	4.8220	0.288	16.723	0.000	4.254	5.390
np.log(sales)	0.2567	0.035	7.436	0.000	0.189	0.325

Omnibus:	84.151	**Durbin-Watson:**	1.860
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	403.831
Skew:	1.507	**Prob(JB):**	2.04e-88
Kurtosis:	9.106	**Cond. No.**	70.0

Exercises: Normality and Intervals

- Regress log(wage) on education
- Determine whether the residuals of this regression are normally distributed
- Generate a summary frame containing the confidence and prediction intervals
- (optional) visualize the confidence and prediction intervals along with the regression line and observations

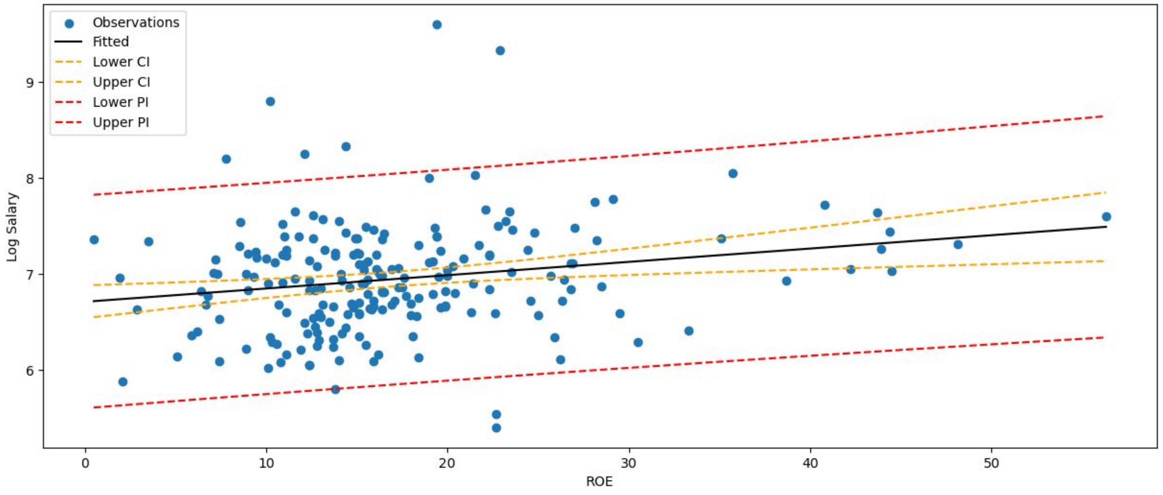


A data.frame with 526 observations on 24 variables:

- **wage:** average hourly earnings
- **educ:** years of education
- **exper:** years potential experience
- **tenure:** years with current employer
- **nonwhite:** =1 if nonwhite
- **female:** =1 if female
- **married:** =1 if married
- **numdep:** number of dependents
- **smsa:** =1 if live in SMSA
- **northcen:** =1 if live in north central U.S
- **south:** =1 if live in southern region
- **west:** =1 if live in western region
- **construc:** =1 if work in construc. indus.
- **ndurman:** =1 if in nondur. manuf. indus.
- **trcommpu:** =1 if in trans, commun, pub ut
- **trade:** =1 if in wholesale or retail
- **services:** =1 if in services indus.
- **profserv:** =1 if in prof. serv. indus.
- **profocc:** =1 if in profess. occupation
- **clerocc:** =1 if in clerical occupation
- **servocc:** =1 if in service occupation
- **lwage:** log(wage)
- **expersq:** exper^2
- **tenursq:** tenure^2

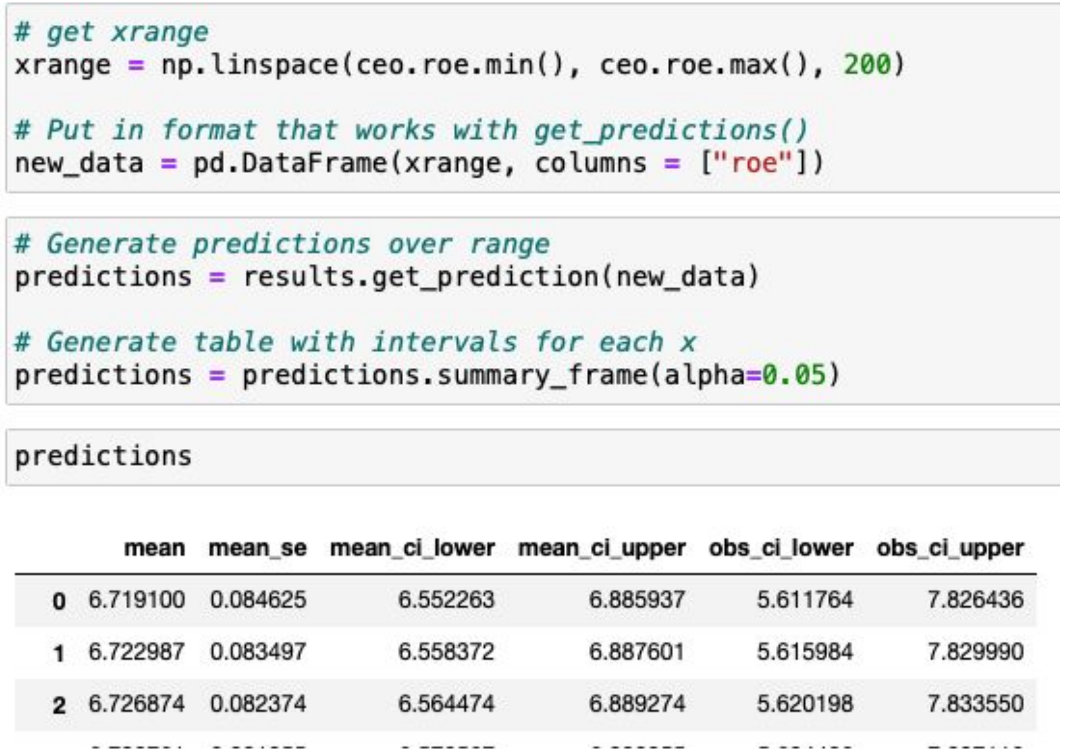
Prediction and Confidence Intervals

- Confidence Intervals: Concerned with estimating the population means for a given value of X
- Prediction Intervals: Predicts the range in which an observation for a given value of X is likely to fall
  - Prediction intervals include the uncertainty inherent to confidence intervals and thus are always wider



Producing Prediction and Confidence Intervals

1. Generate an array of evenly spaced values over the range of X
2. Use results.get_prediction() to get the predicted value for each x in the range
3. Use the predictions.summary_frame() method to get the intervals



```
# get xrange
xrange = np.linspace(ceo.roe.min(), ceo.roe.max(), 200)

# Put in format that works with get_predictions()
new_data = pd.DataFrame(xrange, columns = ["roe"])
```

```
# Generate predictions over range
predictions = results.get_prediction(new_data)

# Generate table with intervals for each x
predictions = predictions.summary_frame(alpha=0.05)
```

```
predictions
```

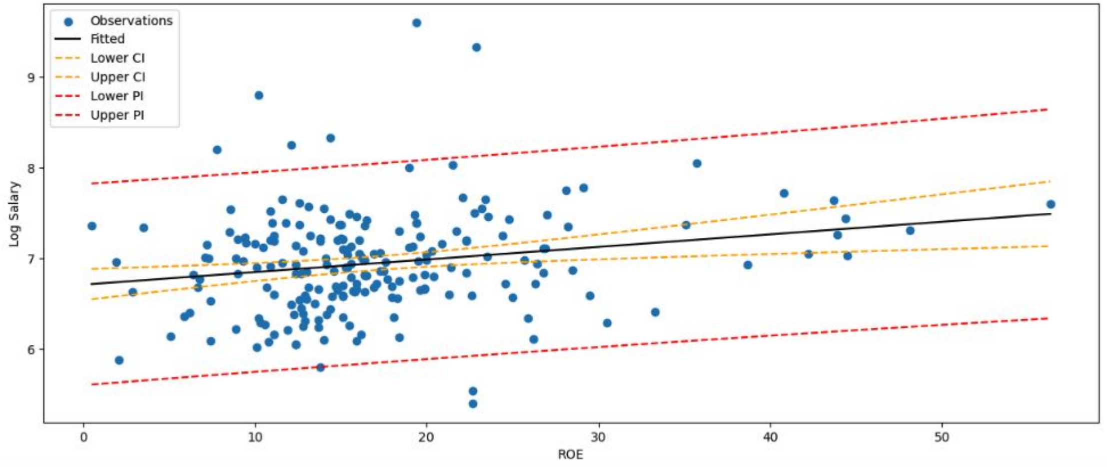
	mean	mean_se	mean_ci_lower	mean_ci_upper	obs_ci_lower	obs_ci_upper
0	6.719100	0.084625	6.552263	6.885937	5.611764	7.826436
1	6.722987	0.083497	6.558372	6.887601	5.615984	7.829990
2	6.726874	0.082374	6.564474	6.889274	5.620198	7.833550

Plotting Prediction and Confidence Intervals

- Four different elements are included in thuis plot:
  - Observed values for each actual x
  - The predicted mean value for each x in xrange (in the "mean" column)
  - The confidence intervals for the regression line (mean_ci_lower and mean_ci_upper)
  - The confidence intervals for the predictions (obs_ci_lower and obs_ci_upper)

```python
plt.figure(figsize = (15, 6))
plt.scatter(ceo.roe, np.log(ceo.salary))
plt.plot(xrange, predictions["mean"], color = "black")

plt.xlabel("ROE")
plt.ylabel("Log Salary")

# confidence Intervals
plt.plot(xrange, predictions["mean_ci_lower"], color = "orange", linestyle = '--')
plt.plot(xrange, predictions["mean_ci_upper"], color = "orange", linestyle = '--')

# prediction Intervals
plt.plot(xrange, predictions["obs_ci_lower"], color = "red", linestyle = '--')
plt.plot(xrange, predictions["obs_ci_upper"], color = "red", linestyle = '--')

# Fun fact - the legend is labelled in the order you draw each plot element!
plt.legend(["Observations", "Fitted", "Lower CI", "Upper CI","Lower PI", "Upper PI"])
```

```
<matplotlib.legend.Legend at 0x7feb092f2fd0>
```

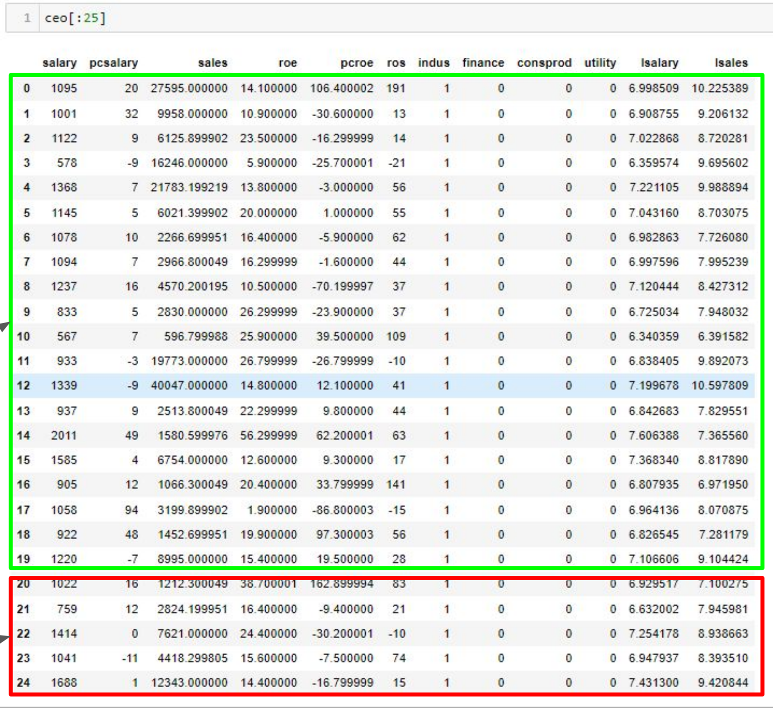


K-Fold Cross Validation

- K-fold cross validation splits our data up into k subsets
- Train our model on k-1 subsets and predict on the one that’s left out
- Repeat over all combinations of subsets
- Helpful to measure overfitting and optimizing model parameters
- On the right I show the first step of 5-fold cross validation where the first 20 observations are used to train and the last 5 are used to test

Train (step 1)

Test (Step 1)

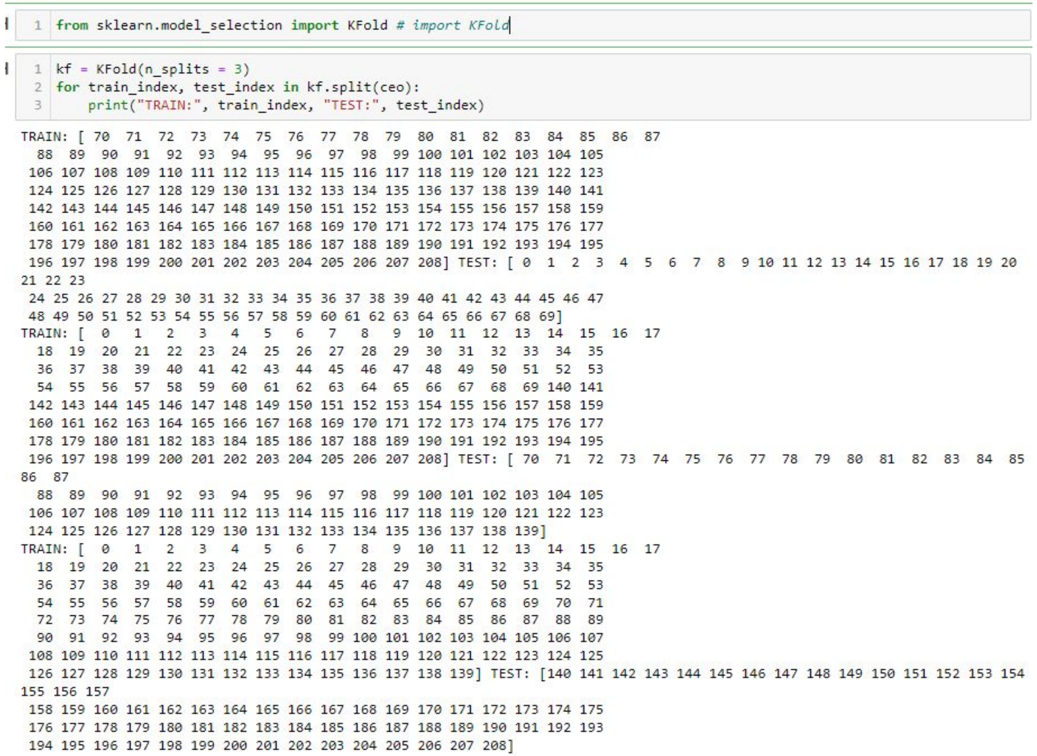


```
ceo[:25]
```

	salary	pcsalary	sales	roe	pcroe	ros	indus	finance	consprod	utility	lsalary	lsales
0	1095	20	27595.000000	14.100000	106.400002	191	1	0	0	0	6.998509	10.225389
1	1001	32	9958.000000	10.900000	-30.600000	13	1	0	0	0	6.908755	9.206132
2	1122	9	6125.899902	23.500000	-16.299999	14	1	0	0	0	7.022868	8.720281
3	578	-9	16246.000000	5.900000	-25.700001	-21	1	0	0	0	6.359574	9.695602
4	1368	7	21783.199219	13.800000	-3.000000	56	1	0	0	0	7.221105	9.988894
5	1145	5	6021.399902	20.000000	1.000000	55	1	0	0	0	7.043160	8.703075
6	1078	10	2266.699951	16.400000	-5.900000	62	1	0	0	0	6.982863	7.726080
7	1094	7	2966.800049	16.299999	-1.600000	44	1	0	0	0	6.997596	7.995239
8	1237	16	4570.200195	10.500000	-70.199997	37	1	0	0	0	7.120444	8.427312
9	833	5	2830.000000	26.299999	-23.900000	37	1	0	0	0	6.725034	7.948032
10	567	7	596.799988	25.900000	39.500000	109	1	0	0	0	6.340359	6.391582
11	933	-3	19773.000000	26.799999	-26.799999	-10	1	0	0	0	6.838405	9.892073
12	1339	-9	40047.000000	14.800000	12.100000	41	1	0	0	0	7.199678	10.597809
13	937	9	2513.800049	22.299999	9.800000	44	1	0	0	0	6.842683	7.829551
14	2011	49	1580.599976	56.299999	62.200001	63	1	0	0	0	7.606388	7.365560
15	1585	4	6754.000000	12.600000	9.300000	17	1	0	0	0	7.368340	8.817890
16	905	12	1066.300049	20.400000	33.799999	141	1	0	0	0	6.807935	6.971950
17	1058	94	3199.899902	1.900000	-86.800003	-15	1	0	0	0	6.964136	8.070875
18	922	48	1452.699951	19.900000	97.300003	56	1	0	0	0	6.826545	7.281179
19	1220	-7	8995.000000	15.400000	19.500000	28	1	0	0	0	7.106606	9.104424
20	1022	16	1212.300049	38.700001	162.899994	83	1	0	0	0	6.929517	7.100275
21	759	12	2824.199951	16.400000	-9.400000	21	1	0	0	0	6.632002	7.945981
22	1414	0	7621.000000	24.400000	-30.200001	-10	1	0	0	0	7.254178	8.938663
23	1041	-11	4418.299805	15.600000	-7.500000	74	1	0	0	0	6.947937	8.393510
24	1688	1	12343.000000	14.400000	-16.799999	15	1	0	0	0	7.431300	9.420844

K-Fold Cross Validation (sklearn)

- Sklearn is the premiers ML library for most models
- Includes tools for testing, tuning, and cross validation
- The kfold submodule will automatically split your data into k subsets
- This is a *generator* function that we can iterate through in a loop
- Can be combined with other compatible functions
- We can also set shuffle = True if we suspect our data isn't arranged randomly



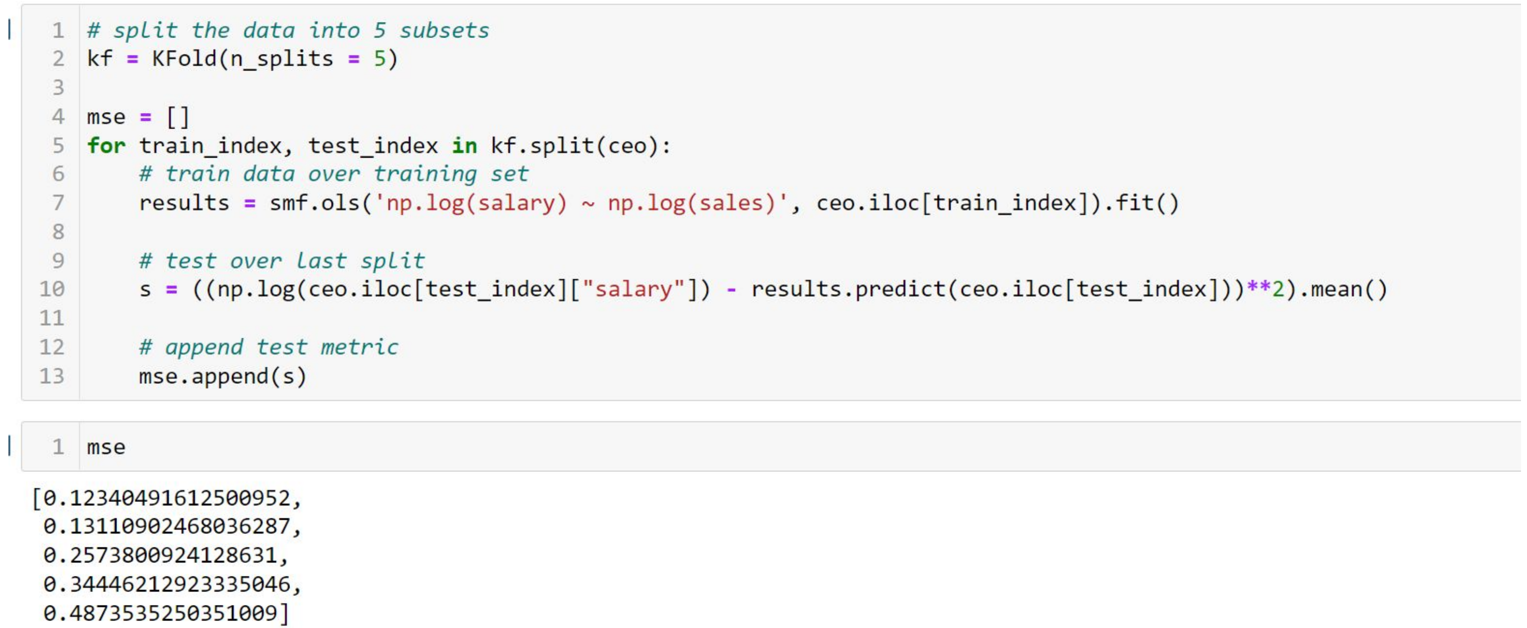
```python
from sklearn.model_selection import KFold # import KFold
```

```python
kf = KFold(n_splits = 3)
for train_index, test_index in kf.split(ceo):
    print("TRAIN:", train_index, "TEST:", test_index)
```

```
TRAIN: [ 70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87
  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105
 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123
 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141
 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195
 196 197 198 199 200 201 202 203 204 205 206 207 208] TEST: [ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
21 22 23
 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47
 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69]
TRAIN: [  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69 140 141
 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159
 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177
 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195
 196 197 198 199 200 201 202 203 204 205 206 207 208] TEST: [ 70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85
86  87
  88  89  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105
 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123
 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139]
TRAIN: [  0   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17
  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35
  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53
  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71
  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89
  90  91  92  93  94  95  96  97  98  99 100 101 102 103 104 105 106 107
 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125
 126 127 128 129 130 131 132 133 134 135 136 137 138 139] TEST: [140 141 142 143 144 145 146 147 148 149 150 151 152 153 154
155 156 157
 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175
 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193
 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208]
```

Manual Cross Validation

- Cross validation can be implemented manually in the following steps:
  - Train model over training indices
  - Make predictions over the test indices
  - Calculate a test metric (MSE, MAPE, RMSE, etc) to evaluate the model's performance against the observed testing values
  - Store the performance for each fold



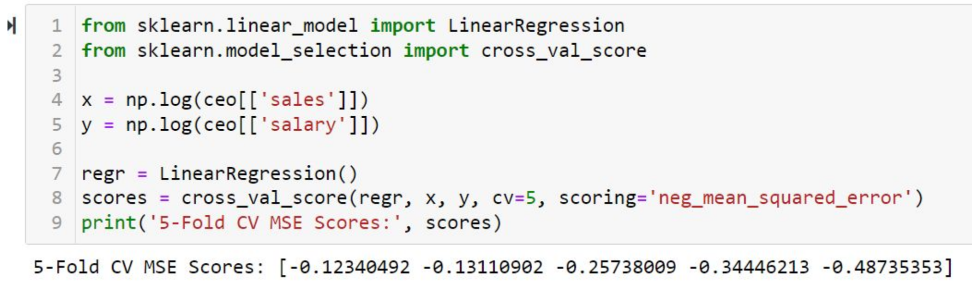
```python
# split the data into 5 subsets
kf = KFold(n_splits = 5)

mse = []
for train_index, test_index in kf.split(ceo):
    # train data over training set
    results = smf.ols('np.log(salary) ~ np.log(sales)', ceo.iloc[train_index]).fit()

    # test over last split
    s = ((np.log(ceo.iloc[test_index]["salary"]) - results.predict(ceo.iloc[test_index]))**2).mean()

    # append test metric
    mse.append(s)
```

```python
mse
```

```
[0.12340491612500952,
 0.13110902468036287,
 0.2573800924128631,
 0.34446212923335046,
 0.4873535250351009]
```

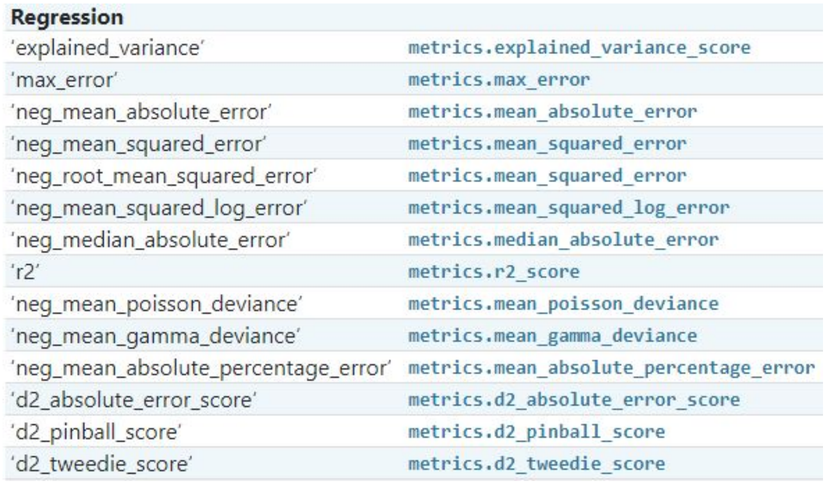
Sklearn Cross Validation

- Cross validation is straightforward in sklearn
- Requires us to use the sklearn regression functions for compatibility
- Cross_val_score has us specify:
  - The model
  - Dependent/independent variables
  - The scoring method
  - Number of folds
- Sklearn seeks to maximize each metric, so this is why the negative is taken of some metrics (like MSE and RMSE)
- Why do we square? Why prefer MSE over RMSE?



```python
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score

x = np.log(ceo[['sales']])
y = np.log(ceo[['salary']])

regr = LinearRegression()
scores = cross_val_score(regr, x, y, cv=5, scoring='neg_mean_squared_error')
print('5-Fold CV MSE Scores:', scores)
```

```
5-Fold CV MSE Scores: [-0.12340492 -0.13110902 -0.25738009 -0.34446213 -0.48735353]
```



Regression	
'explained_variance'	metrics.explained_variance_score
'max_error'	metrics.max_error
'neg_mean_absolute_error'	metrics.mean_absolute_error
'neg_mean_squared_error'	metrics.mean_squared_error
'neg_root_mean_squared_error'	metrics.mean_squared_error
'neg_mean_squared_log_error'	metrics.mean_squared_log_error
'neg_median_absolute_error'	metrics.median_absolute_error
'r2'	metrics.r2_score
'neg_mean_poisson_deviance'	metrics.mean_poisson_deviance
'neg_mean_gamma_deviance'	metrics.mean_gamma_deviance
'neg_mean_absolute_percentage_error'	metrics.mean_absolute_percentage_error
'd2_absolute_error_score'	metrics.d2_absolute_error_score
'd2_pinball_score'	metrics.d2_pinball_score
'd2_tweedie_score'	metrics.d2_tweedie_score