Regression (Continued) III

Today’s Outline

- Cross-Validation
- Bootstrapping
- Transformations
- Boruta and Mallow’s CP

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)

```
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

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

```
1 kf = KFold(n_splits = 3)
2 for train_index, test_index in kf.split(ceo):
3     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

```
# 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)
```

```
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

Bootstrapping

- The standard errors, intervals, and other properties of some estimators make be difficult or impossible to calculate (for example the CI on a lowess smoother)
- Bootstrapping gives us a very powerful way to use resampling to estimate these properties
- The algorithm is straightforward:
  - Get a *representative* sample from your population of size **N**
  - Generate B bootstrap samples of size **N** with replacement from your original sample
  - Calculate your desired statistic over each bootstrap sample
  - Use the distribution of your bootstrap sample statistics to find the moments, confidence intervals, etc. of the original sample statistic
- Also can be used in machine learning to create ensemble models

Manual Bootstrapping

- On the right I show the algorithm for bootstrapping regression coefficients
- This framework can be applied for other statistics and ML models as well
- Here we calculate the percentile confidence interval
  - Note that the percentile method tends to be more vulnerable to bias and skew (not a first choice)

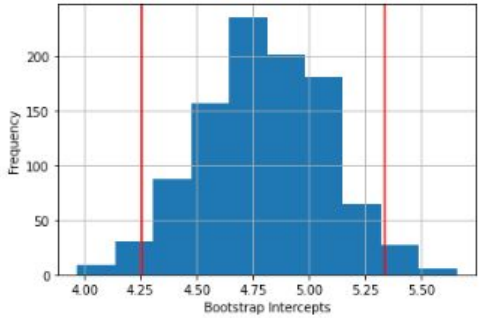
```
# build dataframe to store sample statistics
coefs = pd.DataFrame(columns = ["B0","B1"])

# we will generate 1000 bootstrap samples
for i in range(1000):

    # sample from the data with replacement N times
    sample = ceo.sample(ceo.shape[0] ,replace = True)

    # fit model on bootstrap sample
    results = smf.ols('np.log(salary) ~ np.log(sales)', sample).fit()

    # pull out the bootstrap sample statistics
    b0,b1 = results.params

    # store the bootstrap sample statistics for later use
    coefs = coefs.append({"B0":b0, "B1":b1}, ignore_index = True)

# below I calculate the percentile bootstraps for a 95% confidence interval

# the 97.5 percentile of thebootstrap sample statistics
b0_u, b1_u = coefs.quantile(.975)

# the 2.5 percentile of the bootstrap sample statistics
b0_l, b1_l = coefs.quantile(.025)
```

Plotting Bootstraps

- On the right we plot the bootstrap sample statistics for the slope and intercepts of the regression
- We also include the 95% percentile confidence interval for each statistic

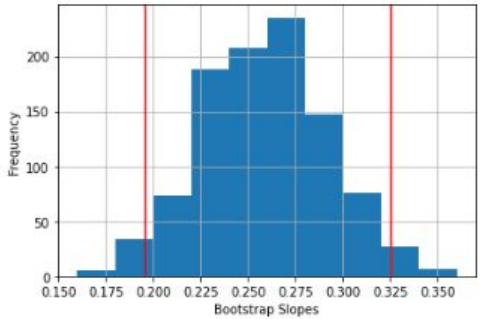
```
1  coefs.B0.hist()
2
3  plt.xlabel("Bootstrap Intercepts")
4  plt.ylabel("Frequency")
5  plt.axvline(b0_u, color = "red")
6  plt.axvline(b0_1, color = "red")
```

```
: <matplotlib.lines.Line2D at 0x1dbfe7b9be0>
```




```
1  coefs.B1.hist()
2  plt.xlabel("Bootstrap Slopes")
3  plt.ylabel("Frequency")
4  plt.axvline(b1_u, color = "red")
5  plt.axvline(b1_1, color = "red")
```

```
: <matplotlib.lines.Line2D at 0x1dbfe977400>
```



Basic Bootstrap Confidence Interval

- Bootstrap standard errors can also be computed using the following basic estimator:

$$SE_{bt}\left(\hat{\theta}\right) = \sqrt{(B-1)^{-1}\sum_{b=1}^{B}\left(\hat{\theta}^{*,b} - \bar{\hat{\theta}}^{*}\right)^2}$$

$$\left[\hat{\theta} - SE_{bt}\left(\hat{\theta}\right)z_{1-\alpha/2}, \hat{\theta} + SE_{bt}\left(\hat{\theta}\right)z_{1-\alpha/2}\right]$$

```
# calculate the bootstrap standard error over the bootstrap statsitics for beta1
se_bt = coefs.B1.std(ddof = 1)

# calculate the basic CI
[b1-1.96*se_bt, b1+1.96*se_bt]
```

```
[0.19157312824957456, 0.3217702550787261]
```

Scipy Stats Bootstrap

```
scipy.stats.bootstrap(data, statistic, *, n_resamples=9999, batch=None,
vectorized=True, paired=False, axis=0, confidence_level=0.95, method='BCa',
random_state=None) #                                                [source]
```

- No convenient function similar to R for bootstrap regression
- However, we can write our own function for the “statistic” argument and use the scipy.stats.bootstrap() function to get similar results
- BCa (bias-corrected CI) is the preferred method for calculating bootstrap confidence intervals and si complicated to implement manually
- BCa can be used with this method

```python
from scipy.stats import bootstrap
```

```python
# Note that you have to write your own functions that will work with
# scipy.stats bootstrap

# The first one takes in x and y, returns beta1
def reg_boot_b1(x,y):

    # bootstrap function gives us a 1d array, need 2d
    x = x.reshape((len(x),1))
    y = y.reshape((len(y),1))
    reg = LinearRegression().fit(x,y)

    # Pull out beta1
    return reg.coef_[0][0]

# The first one takes in x and y, returns beta0
def reg_boot_intercept(x,y):

    # bootstrap function gives us a 1d array, need 2d
    x = x.reshape((len(x),1))
    y = y.reshape((len(y),1))
    reg = LinearRegression().fit(x,y)

    # Pull out beta0
    return reg.intercept_[0]
```

statistic : *callable*

Statistic for which the confidence interval is to be calculated. *statistic* must be a callable that accepts `len(data)` samples as separate arguments and returns the resulting statistic. If *vectorized* is set `True`, *statistic* must also accept a keyword argument *axis* and be vectorized to compute the statistic along the provided *axis*.

Scipy Stats Bootstrap (Continued)

- Function is specified as follows:
  - X, y are taken as arguments
  - We feed in the statistic and confidence intervals we want to calculate
  - Vectorized = False says the axis doesn't matter (for example mean can be calculated over rows or columns)
  - Method can be percentile, basic, or BCa
  - Paired says we need to line up the x and y observations

```
1 from scipy.stats import bootstrap
2 X = np.log(ceo["sales"])
3 Y = np.log(ceo["salary"])
4 res = bootstrap((X,Y), reg_boot, confidence_level=0.95, vectorized = False, method = 'BCa',
5                 paired = True)
```

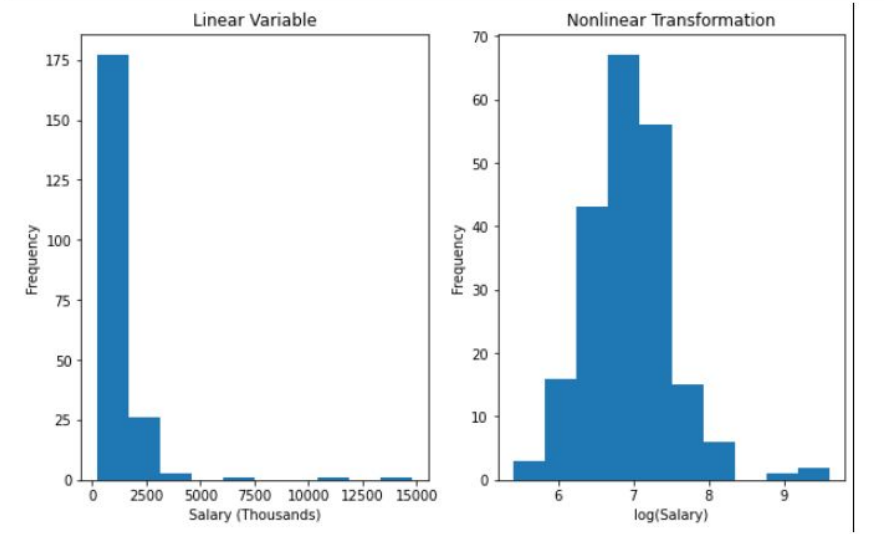
```
1 print(res.confidence_interval)
```

```
ConfidenceInterval(low=0.192185058916601, high=0.3190080132891142)
```

Log-Linear Models

- Log-linear models regress the logarithm of the dependent variable on a linear expression of the dependent variable
- Log stands for the *natural logarithm* which is calculated by default in np.log()
- Log transformation tends to make skewed distributions more ‘normal’
- This transformation might help us meet conditions such as
  - Error normality for hypothesis testing
  - Constant variance
- These models are also convenient to interpret

```
fig, ax = plt.subplots(1, 2, figsize = (10, 6))
ax[0].hist(ceo.salary)
ax[0].set_title("Linear Variable")
ax[0].set_xlabel("Salary (Thousands)")
ax[0].set_ylabel("Frequency")

ax[1].hist(np.log(ceo.salary))
ax[1].set_title("Nonlinear Transformation")
ax[1].set_xlabel("log(Salary)")
ax[1].set_ylabel("Frequency")
plt.show()
```



Log-Linear Model Example

- Transformations can also linearize a relationship between two variables

```
smf.ols('y ~ x', data = sim_data).fit().summary()
```

OLS Regression Results

Dep. Variable:	y	**R-squared:**	0.381
Model:	OLS	**Adj. R-squared:**	0.380
Method:	Least Squares	**F-statistic:**	614.1
Date:	Thu, 19 Oct 2023	**Prob (F-statistic):**	4.95e-106
Time:	14:24:36	**Log-Likelihood:**	-1897.7
No. Observations:	1000	**AIC:**	3799.
Df Residuals:	998	**BIC:**	3809.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	1.3362	0.073	18.278	0.000	1.193	1.480
x	1.2626	0.051	24.780	0.000	1.163	1.363

Omnibus:	742.125	**Durbin-Watson:**	1.858
Prob(Omnibus):	0.000	**Jarque-Bera (JB):**	22275.648
Skew:	3.051	**Prob(JB):**	0.00
Kurtosis:	25.302	**Cond. No.**	2.68

$$log(y) = .2 + .5x + e$$

```
smf.ols('np.log(y) ~ x', data = sim_data).fit().summary()
```

OLS Regression Results

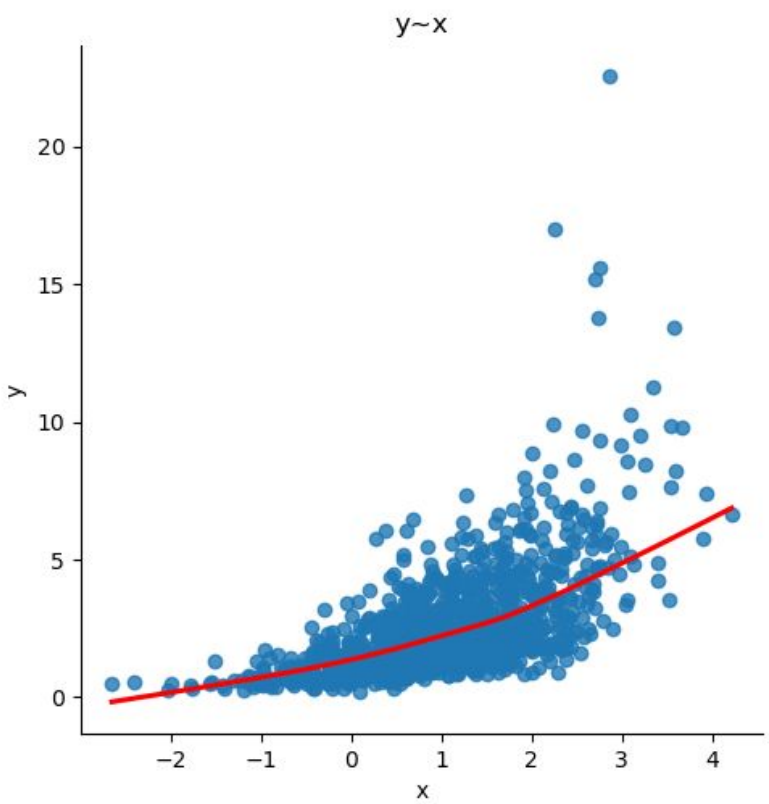
Dep. Variable:	np.log(y)	**R-squared:**	0.505
Model:	OLS	**Adj. R-squared:**	0.504
Method:	Least Squares	**F-statistic:**	1018.
Date:	Thu, 19 Oct 2023	**Prob (F-statistic):**	1.63e-154
Time:	14:24:50	**Log-Likelihood:**	-717.64
No. Observations:	1000	**AIC:**	1439.
Df Residuals:	998	**BIC:**	1449.
Df Model:	1		
Covariance Type:	nonrobust		

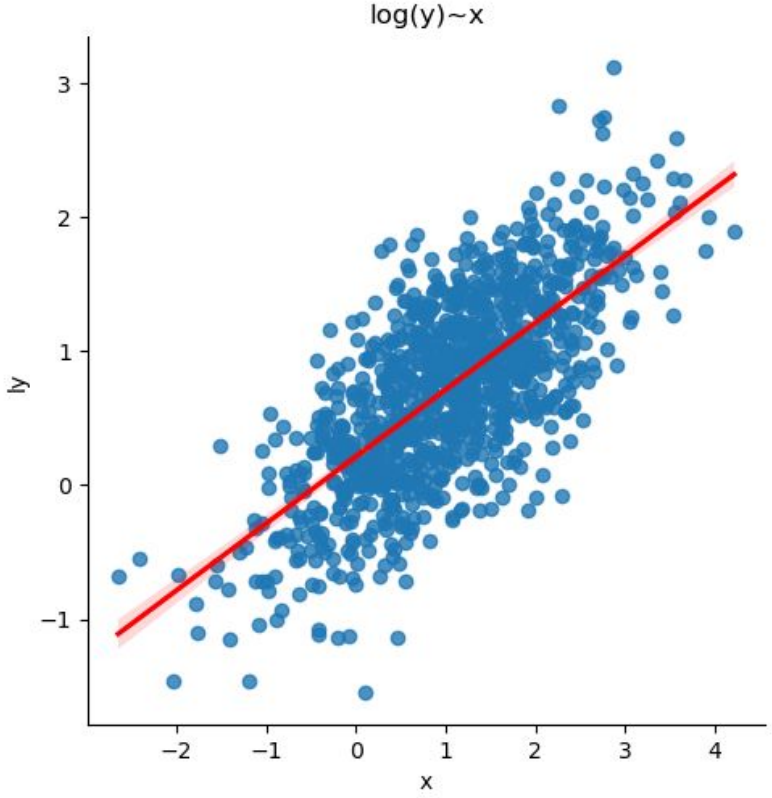
	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	0.2127	0.022	9.472	0.000	0.169	0.257
x	0.4994	0.016	31.901	0.000	0.469	0.530

Omnibus:	1.041	**Durbin-Watson:**	1.876
Prob(Omnibus):	0.594	**Jarque-Bera (JB):**	1.066
Skew:	-0.078	**Prob(JB):**	0.587
Kurtosis:	2.965	**Cond. No.**	2.68

Log-Linear Model Example (Plotted)

- Transformations can also linearize a relationship between two variables





Log-Log Model Example

$$log(y) = .2 + .5 * log(x) + e$$

```
smf.ols('np.log(y) ~ x', data = sim_data).fit().summary()
```

OLS Regression Results

Dep. Variable:	np.log(y)	**R-squared:**	0.333
Model:	OLS	**Adj. R-squared:**	0.332
Method:	Least Squares	**F-statistic:**	417.3
Date:	Thu, 19 Oct 2023	**Prob (F-statistic):**	1.54e-75
Time:	15:09:56	**Log-Likelihood:**	-663.46
No. Observations:	837	**AIC:**	1331.
Df Residuals:	835	**BIC:**	1340.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	-0.4239	0.035	-12.117	0.000	-0.493	-0.355
x	0.4749	0.023	20.428	0.000	0.429	0.521

Omnibus:	9.767	**Durbin-Watson:**	2.026
Prob(Omnibus):	0.008	**Jarque-Bera (JB):**	9.773
Skew:	-0.246	**Prob(JB):**	0.00755
Kurtosis:	3.197	**Cond. No.**	3.84

```
smf.ols('np.log(y) ~ np.log(x)', data = sim_data).fit().summary()
```

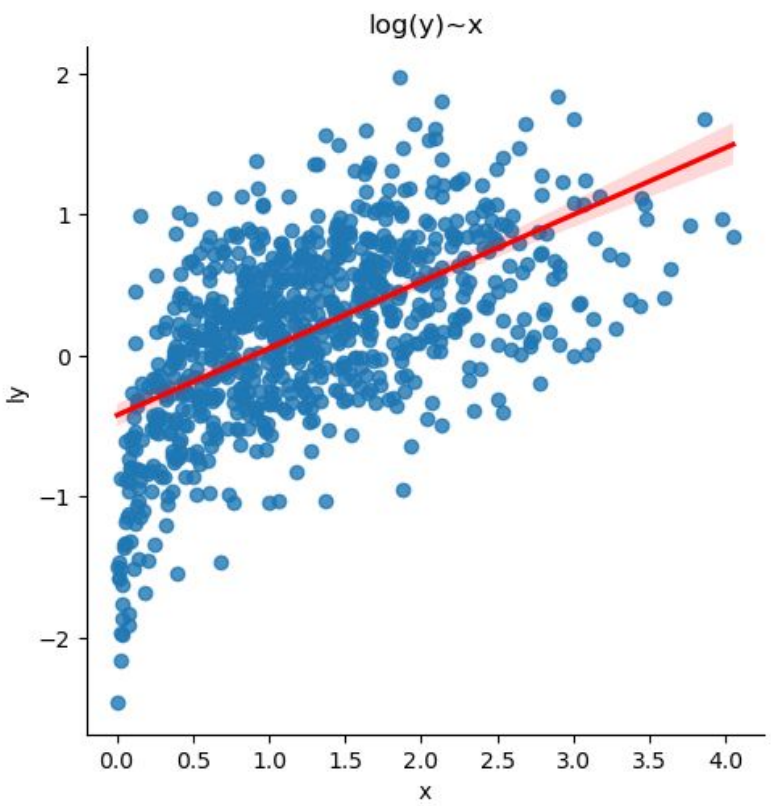
OLS Regression Results

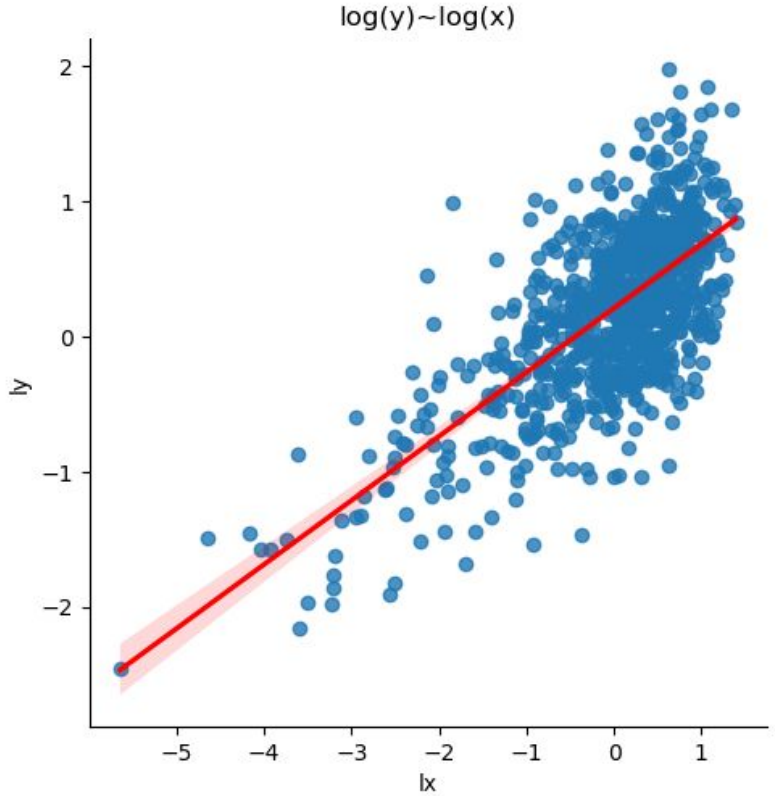
Dep. Variable:	np.log(y)	**R-squared:**	0.465
Model:	OLS	**Adj. R-squared:**	0.465
Method:	Least Squares	**F-statistic:**	726.9
Date:	Thu, 19 Oct 2023	**Prob (F-statistic):**	1.15e-115
Time:	15:09:57	**Log-Likelihood:**	-571.01
No. Observations:	837	**AIC:**	1146.
Df Residuals:	835	**BIC:**	1155.
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	0.2068	0.017	12.468	0.000	0.174	0.239
np.log(x)	0.4738	0.018	26.961	0.000	0.439	0.508

Omnibus:	0.288	**Durbin-Watson:**	2.092
Prob(Omnibus):	0.866	**Jarque-Bera (JB):**	0.192
Skew:	0.024	**Prob(JB):**	0.908
Kurtosis:	3.057	**Cond. No.**	1.08

Log-Log Model Example (Plotted)





Log-Linear Statsmodels

- Non-linear regressions can be fit using statsmodels
- Add np.log(variable) for each variable you want to take the log of in the model
- Here log(salary) is the dependent variable
- I can then print out a summary of the regression results using the result.summary() method

$$\ln(salary) = \beta_0 + \beta_1 sales + u$$

```
# log-linear model
reg1 = smf.ols('np.log(salary) ~ sales', data = ceo)
results1 = reg1.fit()

# look at parameters
results1.summary()
```

OLS Regression Results

Dep. Variable:	np.log(salary)	R-squared:	0.079
Model:	OLS	Adj. R-squared:	0.075
Method:	Least Squares	F-statistic:	17.79
Date:	Wed, 05 Oct 2022	Prob (F-statistic):	3.70e-05
Time:	15:55:54	Log-Likelihood:	-168.63
No. Observations:	209	AIC:	341.3
Df Residuals:	207	BIC:	347.9
Df Model:	1		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	6.8467	0.045	152.138	0.000	6.758	6.935
sales	1.498e-05	3.55e-06	4.217	0.000	7.98e-06	2.2e-05

Omnibus:	57.347	Durbin-Watson:	1.893
Prob(Omnibus):	0.000	Jarque-Bera (JB):	205.912
Skew:	1.063	Prob(JB):	1.94e-45
Kurtosis:	7.373	Cond. No.	1.51e+04

Log-Log Statsmodels

- Add np.log(variable) for each variable you want to take the log of in the model
- Here log(salary) is the dependent variable
- log (sales) is the predictor
- I can then print out a summary of the regression results using the result.summary() method

$$\ln(salary) = \beta_0 + \beta_1 \ln(sales) + u$$

```
# log-log model
reg2 = smf.ols('np.log(salary) ~ np.log(sales)', data = ceo)
results2 = reg2.fit()

results2.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:	Wed, 05 Oct 2022	**Prob (F-statistic):**	2.70e-12
Time:	15:56:28	**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

Interpreting log-linear and log-log coefficients

- The slope associated with a given independent variable log linear model is interpreted as a *semi-elasticity*
- The slope associated with a given independent variable log-log model is the *elasticity*

We interpret this model using semi-elasticity, such that a 1 unit move in x is associated with a $(100 * \beta_1)\%$ rise in y (approximately).

```
# Log-linear models
r1_b1 = results1.params[1]
print("A one unit increase in sales is associated with a", str(r1_b1*100)+"%", "increase in ceo salary.")
```

```
A one unit increase in sales is associated with a 0.0014982498318048918% increase in ceo salary.
```

We interpret this model using elasticity, such that a 1% unit move in x is associated with a $\beta_1\%$ rise in y.

```
# Log-log models
r2_b1 = results2.params[1]
```

```
print("A one percent increase in sales is associated with a", str(r2_b1)+"%", "increase in ceo salary.")
```

```
A one percent increase in sales is associated with a 0.2566716916641503% increase in ceo salary.
```

Exercises Bootstrap and Cross-Validation

- Fit the following two regressions
  - log(wage) ~educ
  - log(wage) ~exper
- Use 5-fold cross validation to determine which model produces the best out of sample predictions using “neg_mean_squared_error”
- Create a bootstrap confidence interval for the slope parameter (b1) in the *log(wage) ~ educ* regression

Box-Cox Power Transform

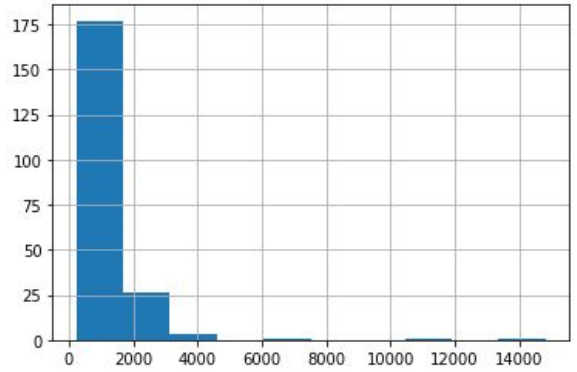
- Like a log transformation, the box-cox transformation can help us meet normality assumptions
- Simple to implement manually
- Manual implementation does not find the best lambda

$$T_{BC}(x,\lambda) = x^{(\lambda)} = \begin{cases} \frac{x^{\lambda}-1}{\lambda} & \text{when } \lambda \neq 0; \\ log_e x & \text{when } \lambda = 0. \end{cases}$$

```
1  def box_cox(x, l = 0):
2      if l == 0:
3          bc = np.log(x)
4      else:
5          bc = (x**l-1)/l
6
7      return bc
8
```

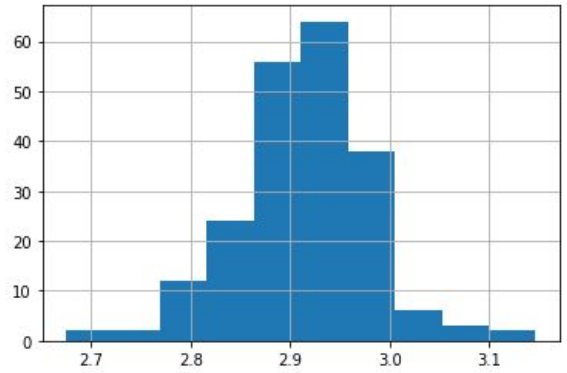
```
1  ceo.salary.hist()
```

```
<AxesSubplot:>
```




```
1  box_cox(ceo.salary, -.3).hist()
```

```
<AxesSubplot:>
```

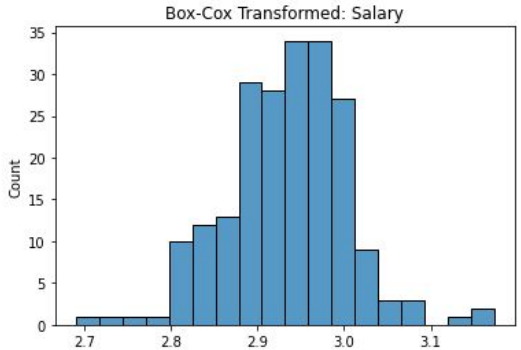


Scipy Box-Cox

- The scipy.stats.boxcox() function finds the value of lambda that maximizes the log-likelihood function
- Returns the desired lambda and the transformed data

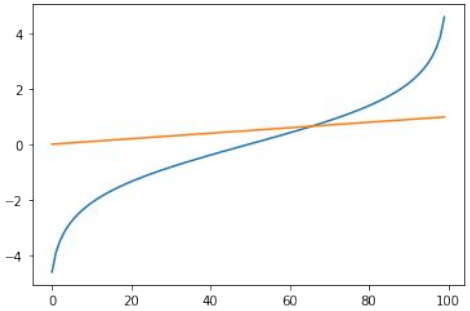
```
1 bc_salary, lambda_salary = stats.boxcox(ceo["salary"])
2 print(lambda_salary)
3 sns.histplot(bc_salary)
4 plt.title("Box-Cox Transformed: Salary")
5 plt.show()
```

```
-0.2970139553908451
```



Restricted Range Transformation: Logit

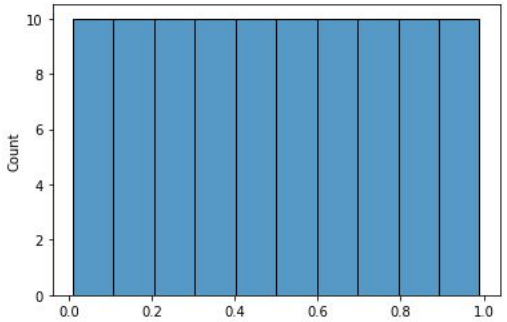
- The arcsin square root transformation (shown in 430 notes) and logit transformation are used on variables with ranges restricted between 0 and 1
- These variables often have values that cluster at 0 and 1
- Transformation stretches out these values towards the ends and makes relationships easier to understand
- Also stabilizes the variance and tends to make the data more normal



$$T_{logit} = \text{logit}(x) = \ln\left(\frac{x}{1-x}\right)$$

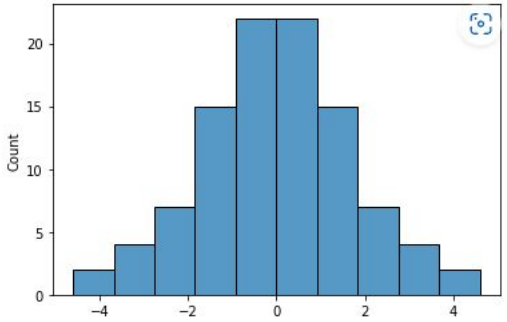
```
1 original = np.linspace(0.01, .99, 100)
2 sns.histplot(original, bins = 10)
```

```
<AxesSubplot:ylabel='Count'>
```




```
1 transformed = np.log(original/(1-original))
2 sns.histplot(transformed, bins = 10)
```

```
<AxesSubplot:ylabel='Count'>
```



Boruta

- BorutaShap uses random forest to measure variable importance
- The runtime of Borutashap scales linearly with the number of observations, and thus can take a very long time to run with large datasets

```
# susbet non-redundant predictors (dont keep salary or varaibales and their log versions)
boruta_data = ceo[["salary", "roe", "ros", "indus", "finance",
                   "consprod", "utility", "lsales"]].copy()
```

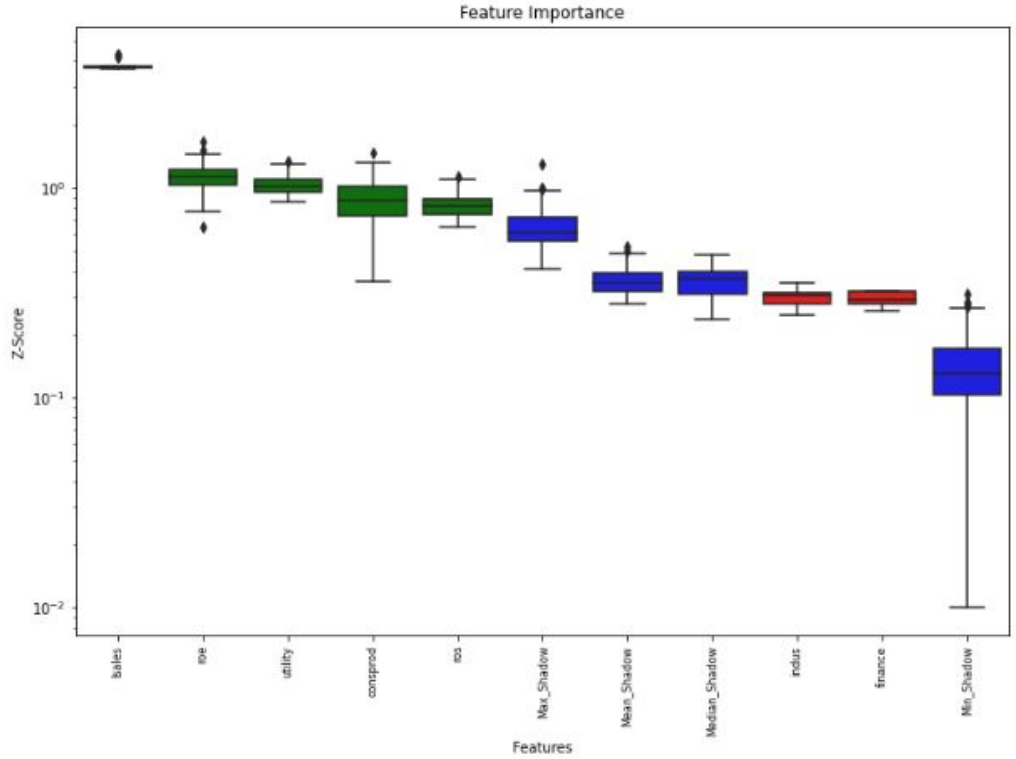
```
from BorutaShap import BorutaShap
x = boruta_data.iloc[:, 1:]
y = np.log(boruta_data['salary'])
```

```
# if model is not specified in BroutaShap(): default = random forest (just like R and BorutaPy)
Feature_Selector = BorutaShap(importance_measure='shap', classification=False)
Feature_Selector.fit(X=x, y=y, n_trials=50, random_state=0)
Feature_Selector.plot(which_features='all')
```

```
100% 50/50 [00:24<00:00, 2.06it/s]
```

Boruta

```
5 attributes confirmed important: ['consprod', 'utility', 'lsales', 'ros', 'roe']
2 attributes confirmed unimportant: ['finance', 'indus']
0 tentative attributes remains: []
```




```
1 # Returns a subset of the original data with the selected features
2 Feature_Selector.Subset()
```

]:

	utility	roe	consprod	lsales	ros
0	0	14.1	0	10.225389	191
1	0	10.9	0	9.206132	13
2	0	23.5	0	8.720281	14
3	0	5.9	0	9.695602	-21
4	0	13.8	0	9.988894	56

Mallows CP

- Mallows CP is another metric that we can use to tell us the best subset of features to include in our model
- The mallow() function from RegscorePy allows us to make this calculation by plugging in:
  - The observed values (y)
  - The fitted values from the full regression (y_pred)
  - The fitted values from a regression on a feature subset (y_sub)
  - The number of parameters in the full regression (k)
  - The number of parameters in the subset regression (p)

```
from RegscorePy import mallow
```

```
model = smf.ols(formula='np.log(salary) ~ roe + np.log(sales) + consprod +ros + utility + finance + utility', data=ceo)
results = model.fit()
y = np.log(ceo['salary'])
y_pred=results.fittedvalues

# You need to run each sub regression individually, and get the score for each subset
# Using subset size =1
mr_sub = smf.ols(formula='np.log(salary) ~ lsales +consprod', data=ceo)
mr_sub_fit = mr_sub.fit()
y_sub=mr_sub_fit.fittedvalues

k = 8 # number of parameters in orginal model (includes y-intercept)
p = 3 # number of parameters in the subset model (includes y-intercept)

mallow.mallow(y, y_pred,y_sub, k, p)
```

```
27.255410002727672
```

Subsets Loop

- Every subset of features needs to be run individually
- This can be done in a for loop
- The itertools.combinations() function will generate all possible subsets of an iterable object (like a list)
- Then we can combine this with the code in the previous slide to calculate Mallow's CP on all subsets
- Note that we have to cut out the empty set in the full code

```
1 import itertools
2 for L in range(len(["A", "B", "C"])+1):
3     for subset in itertools.combinations(["A", "B", "C"], L):
4         print(subset)
```

```
()
('A',)
('B',)
('C',)
('A', 'B')
('A', 'C')
('B', 'C')
('A', 'B', 'C')
```

Loops through all possible sized subsets from 1:n_columns

Returns all possible subsets of size L

Mallows CP Loop

```
subdat = ceo[['salary', 'roe','ros', 'indus','finance', 'consprod', 'utility', 'lsales']].copy()
```

```
import itertools

# get the base model, y and its fitted values
model = smf.ols(formula='np.log(salary) ~ roe + lsales + consprod +ros + utility + finance + utility', data=ceo)
results = model.fit()
y = np.log(ceo['salary'])
y_pred=results.fittedvalues


storage_cp = pd.DataFrame(columns = ["Variables", "CP"])
k = 8 # number of parameters in orginal model (includes y-intercept)

for L in range(1, len(subdat.columns[1:]) + 1):
    for subset in itertools.combinations(subdat.columns[1:], L):

        # join the strings in the data together
        formula1 = 'np.log(salary)~'+'+'.join(subset)

        # get the cp
        results = smf.ols(formula=formula1, data = ceo).fit()
        y_sub = results.fittedvalues
        p = len(subset)+1 # number of parameters in the subset model (includes y-intercept)

        cp = mallow.mallow(y, y_pred,y_sub, k, p)

        # add to the dataframe
        storage_cp = storage_cp.append({'Variables': subset, 'CP': cp}, ignore_index = True)
```

Exercises Boruta and Mallow's CP

- Use the boruta algorithm to select the optimal subset of variables
- Use the following to get started:

```
boruta_data = data[["wage", "exper", "tenure", "female", "nonwhite"]].copy()
```

```
from BorutaShap import BorutaShap
x = boruta_data.iloc[:, 1:]
y = np.log(boruta_data['wage'])
```

- Calculate Mallow's CP for a regression using the full model above and a sub-regression containing only education as a regressor