mlreg_concrete

September 10, 2023

1 Regression on Concrete Data

1.1 Introduction

Concrete is the most important material in civil engineering. The dataset has 9 attributes including 8 quantitative input variables, and 1 quantitative output variable. The dataset can be downloaded and viewed at: https://archive.ics.uci.edu/ml/datasets/concrete+compressive+strength

The compressive strength of concrete determines its quality, and is tested by a standard crushing test on a concrete cylinder. Concrete strength is also considered a key factor in obtaining the desired durability. But testing for strength can take 28 days, which is very long. Our aim is to use Machine Learning to reduce this effort and be able to predict the composition of raw materials for good compressive strength. This is an example of *Surrogate Modeling*, an important approach for combining Machine Learning with scientific research.

The features are:

- Cement: a substance used for construction that hardens to other materials to bind them together.
- Slag: Mixture of metal oxides and silicon dioxide.
- Fly ash: coal combustion product that is composed of the particulates that are driven out of coal-fired boilers together with the flue gases.
- Water: used to form a thick paste.
- Superplasticizer: used in making high-strength concrete.
- Coaseseaggregate: prices of rocks obtained from ground deposits.
- Fine aggregate: the size of aggregate smaller than 4.75mm.
- Age: Rate of gain of strength is faster to start with and the rate gets reduced with age. csMPa: Measurement unit of concrete strength. This variable, present in the original dataste, is not used here.
- Air entrainement: a categorical variable, signalling the use of air entraienment, known to have a beneficial effect in resisting the damage caused to concrete by the freezing/thawing cycles, but a negative effect on the compressive strength of concrete.

The output is:

- Concrete compressive strength: a value in MPa

1.2 Simple Linear Regression

1.3 Preliminaries

[1]:
```python
import pandas as pd
#con = pd.read_csv('concrete_data.csv')
con = pd.read_csv('ConcreteStrength.csv', sep=';', decimal=",")
con.rename(columns={'Fly ash': 'FlyAsh', 'Coarse Aggr.': "CoarseAgg",
                    'Fine Aggr.': 'FineAgg', 'Air Entrainment': 'AirEntrain',
                    'Compressive Strength (28-day)(Mpa)': 'Strength'},␣
 ↪inplace=True)
con['AirEntrain'] = con['AirEntrain'].astype('category')
con.head()
```

[1]:
```
   No  Cement   Slag  FlyAsh  Water    SP  CoarseAgg  FineAgg AirEntrain  \
0   1   273.0   82.0   105.0  210.0   9.0      904.0    680.0         No
1   2   163.0  149.0   191.0  180.0  12.0      843.0    746.0        Yes
2   3   162.0  148.0   191.0  179.0  16.0      840.0    743.0        Yes
3   4   162.0  148.0   190.0  179.0  19.0      838.0    741.0         No
4   5   154.0  112.0   144.0  220.0  10.0      923.0    658.0         No

   Strength
0    34.990
1    32.272
2    35.450
3    42.080
4    26.820
```

1.4 Linear regression with a single explanatory variable

There are many ways to do linear regression in Python. We have already seen the Statsmodels library, so we will continue to use it here. It has much more functionality than we need, but it provides nicely-formatted output.

The method we will use to create linear regression models in the Statsmodels library is `OLS()`. OLS stands for "ordinary least squares", which means the algorithm finds the best fit line by minimizing the squared residuals (this is "least squares"). The "ordinary" part of the name gives us the sense that the type of linear regression we are seeing here is just the tip of the methodological iceberg. There is a whole world of non-ordinary regression techniques out there intended to address this or that methodological problem or circumstance. These will be seen in a later example.

1.4.1 Preparing the data

Recall the general format of the linear regression equation:

$$Y = \beta_0 + \beta_1 X_1 + ... + \beta_p X_p,$$

where Y is the value of the response variable and X_i is the value of the explanatory variable(s).

If we think about this equation in matrix terms, we see that Y is a 1-dimensional matrix: it is just a single column (or array or vector) of numbers. In our case, this vector corresponds to the compressive strength of different batches of concrete measured in megapascals. The right-hand side of the equation is actually a 2-dimensional matrix: there is one column for our X variable and another column for the constant.

Creating a linear regression model in Statsmodels requires the following steps: 1. Import the Statsmodels library 2. Define Y and X matrices. This is optional, but it keeps the `OLS()` call easier to read 3. Add a constant column to the X matrix 4. Call `OLS()` to define the model 5. Call `fit()` to actually estimate the model parameters using the data set (fit the line) 6. Display the results

Let's start with the first three steps:

[2]:
```
import statsmodels.api as sm
Y = con['Strength']
X = con['FlyAsh']
X.head()
```

[2]:
```
0    105.0
1    191.0
2    191.0
3    190.0
4    144.0
Name: FlyAsh, dtype: float64
```

We see above that X is a single column of numbers (amount of fly ash in each batch of concrete). The numbers on the left are just the Python index (every row in a Python array has a row number, or index).

1.4.2 Adding a column for the constant

We can add another column for the regression constant using Statsmodels `add_constant()` method:

[3]:
```
X = sm.add_constant(X)
X.head()
```

[3]:

	const	FlyAsh
0	1.0	105.0
1	1.0	191.0
2	1.0	191.0
3	1.0	190.0
4	1.0	144.0

Notice the difference: the X matrix has been augmented with a column of 1s called "const". To see why, recall the point of linear regression: to use data to "learn" the parameters of the best-fit line and use the parameters to make predictions. The parameters of a line are its y-intercept and slope. Once we have the y-intercept and slope (β_0 and β_1 in the equation above or b and m in grade 9 math), we can multiply them by the data in the X matrix to get a prediction for Y.

Written out in words for the first row of our data, we get:

Concrete strength estimate = β_0 x 1 + β_1 x 105.0

The “const” column simply provides a placeholder—a bunch of 1’s to multiply the constant by. So now we understand why we have to run `add_constant()`.

1.4.3 Running the model

[4]:
```python
model = sm.OLS(Y, X, missing='drop')

model_result = model.fit()
model_result.summary()
```

[4]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:               Strength   R-squared:                       0.165
Model:                            OLS   Adj. R-squared:                  0.157
Method:                 Least Squares   F-statistic:                     19.98
Date:                Sun, 10 Sep 2023   Prob (F-statistic):           2.05e-05
Time:                        12:21:45   Log-Likelihood:                -365.58
No. Observations:                 103   AIC:                             735.2
Df Residuals:                     101   BIC:                             740.4
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const         26.2764      1.691     15.543      0.000      22.923      29.630
FlyAsh         0.0440      0.010      4.470      0.000       0.024       0.064
==============================================================================
Omnibus:                        5.741   Durbin-Watson:                   1.098
Prob(Omnibus):                  0.057   Jarque-Bera (JB):                2.716
Skew:                           0.064   Prob(JB):                        0.257
Kurtosis:                       2.215   Cond. No.                         346.
==============================================================================

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

This output look very similar to what we have seen before.

```
<B>Note:</B>
There is missing data here, so the <CODE>missing='drop'</CODE> argument above is required.  Mi
```

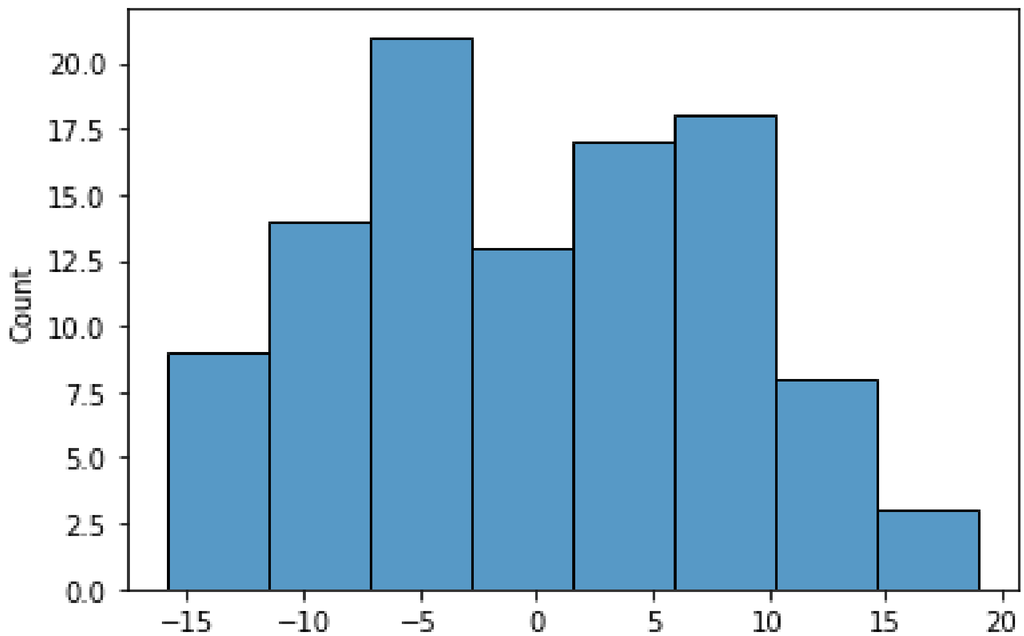
1.5 Regression diagnostics

Like R, Statsmodels exposes the residuals. That is, keeps an array containing the difference between the observed values Y and the values predicted by the linear model. A fundamental assumption is that the residuals (or "errors") are random: some big, some some small, some positive, some negative, but overall the errors should be normally distributed with mean zero. Anything other than normally distributed residuals indicates a serious problem with the linear model.

1.6 Histogram of residuals

Plotting residuals in Seaborn is straightforward: we simply pass the `histplot()` function the array of residuals from the regression model.

[5]:
```
import seaborn as sns
sns.histplot(model_result.resid);
```



A slightly more useful approach for assessing normality is to compare the kernel density estimate with the curve for the corresponding normal curve. To do this, we generate the normal curve that has the same mean and standard deviation as our observed residual and plot it on top of our residual.

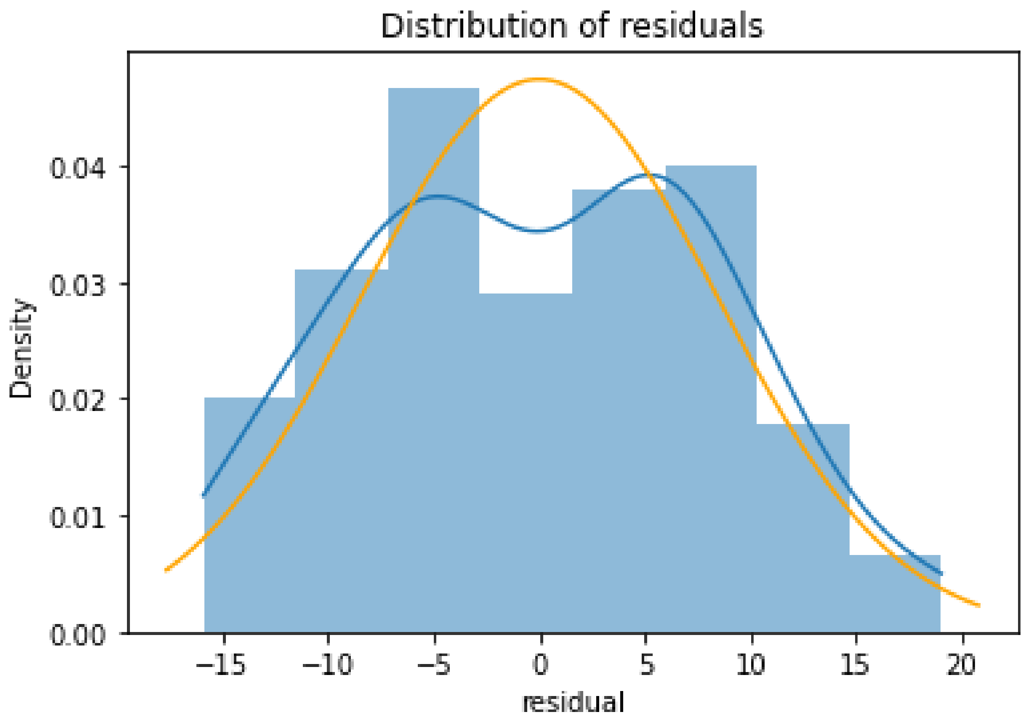
We use a Python trick to assign two values at once: the `fit()` function returns both the mean and the standard deviation of the best-fit normal distribution.

[6]:
```
from scipy import stats
mu, std = stats.norm.fit(model_result.resid)
mu, std
```

[6]:
```
(4.415022824140428e-15, 8.418278511304978)
```

We can now re-plot the residuals as a kernel density plot and overlay the normal curve with the same mean and standard deviation:

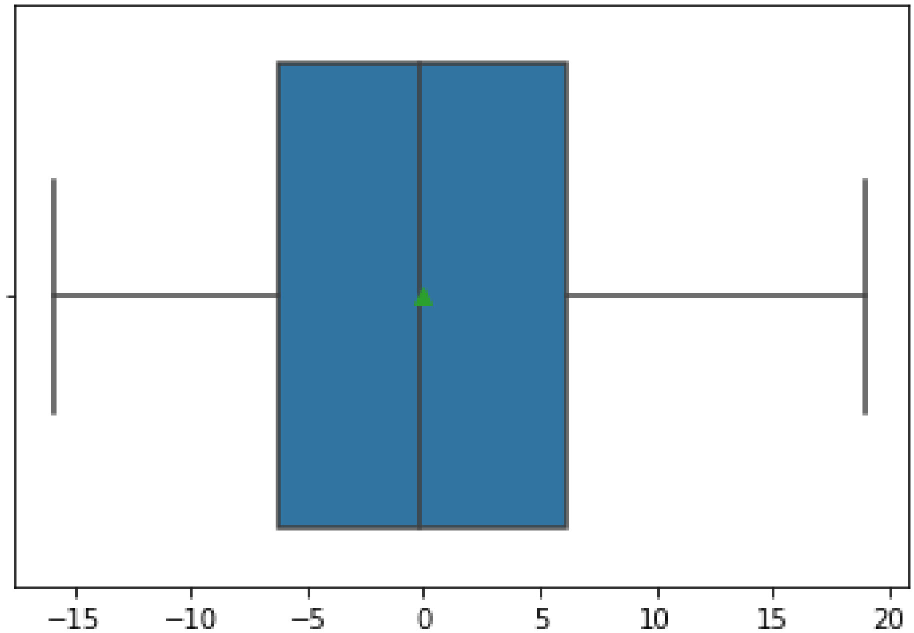
[7]:
```
import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
# plot the residuals
sns.histplot(x=model_result.resid, ax=ax, stat="density", linewidth=0, kde=True)
ax.set(title="Distribution of residuals", xlabel="residual")

# plot corresponding normal curve
xmin, xmax = plt.xlim() # the maximum x values from the histogram above
x = np.linspace(xmin, xmax, 100) # generate some x values
p = stats.norm.pdf(x, mu, std) # calculate the y values for the normal curve
sns.lineplot(x=x, y=p, color="orange", ax=ax)
plt.show()
```



1.7 Boxplot of residuals

A boxplot is often better when the residuals are highly non-normal. Here we see a reasonable distribution with the mean close to the median (indicating symmetry).

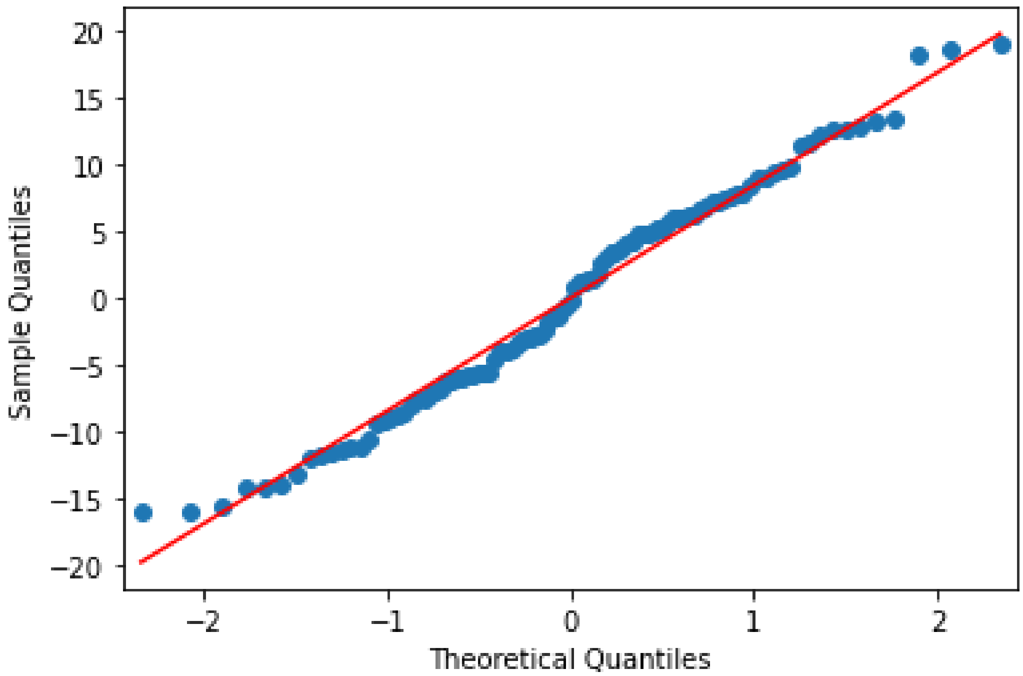
[8]:
```
sns.boxplot(x=model_result.resid, showmeans=True);
```



1.8 Q-Q plot

A Q-Q plot is a bit more specialized than a histogram or boxplot, so the easiest thing is to use the regression diagnostic plots provided by Statsmodels. These plots are not as attractive as the Seaborn plots, but they are intended primarily for the data analyst.

[9]:
```
sm.qqplot(model_result.resid, line='s');
```

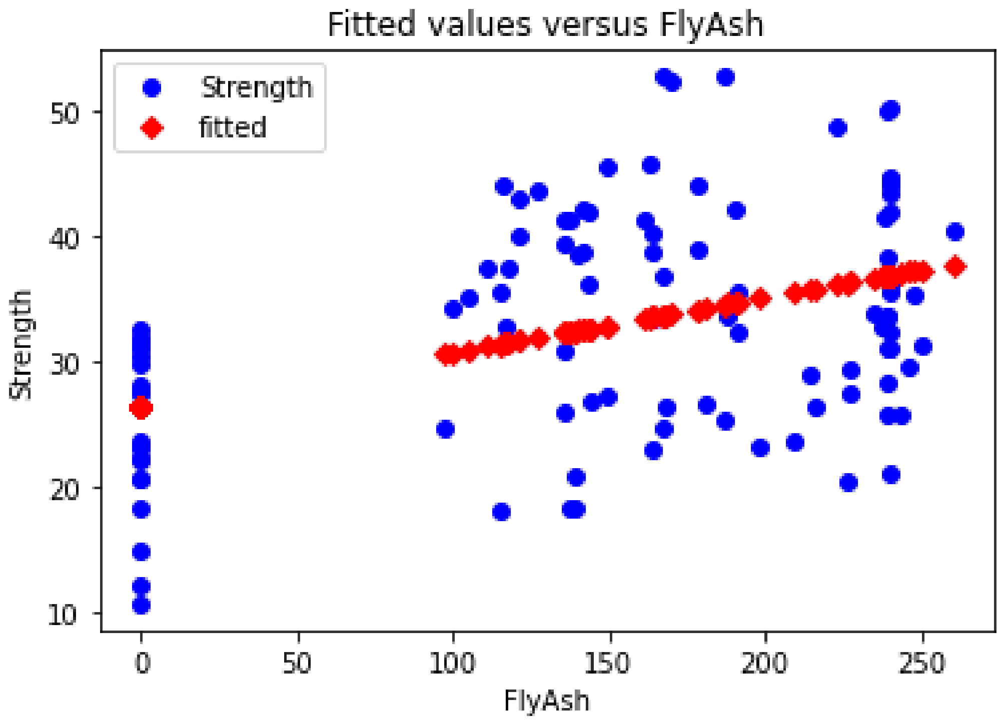


1.9 Fit plot

A fit plot shows predicted values of the response variable versus actual values of Y. If the linear regression model is perfect, the predicted values will exactly equal the observed values and all the data points in a predicted versus actual scatterplot will fall on the 45° diagonal.

The fit plot provided by Statsmodels gives a rough sense of the quality of the model. Since the R^2 of this model is only 0.01, it should come as no surprise that the fitted model is not particularly good.

[10]:
```
sm.graphics.plot_fit(model_result,1, vlines=False);
```



1.10 Fit plot in seaborn

As in R, creating a better fit plot is a bit more work. The central issue is that the observed and predicted axis must be identical for the reference line to be 45°. This can be done as follows:

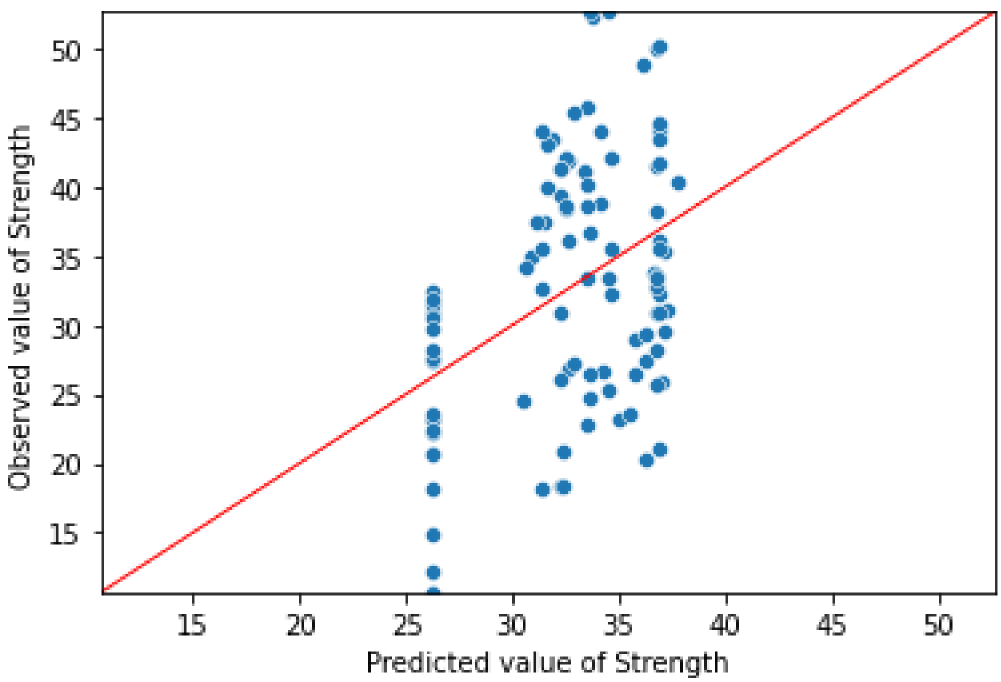
1. Determine the min and max values for the observed values of Y
2. Predict values of Y
3. Create a plot showing the observed versus predicted values of Y. Save this to an object (in this case `ax`)
4. Modify the chart object so that the two axes share the same minimum and maximum values
5. Generate data on a 45° line and add the reference line to the plot

[11]:
```
model_result.fittedvalues
```

[11]:
```
0      30.901424
1      34.689565
2      34.689565
3      34.645517
4      32.619302
         …
98     36.808282
99     36.843520
100    36.830306
101    36.843520
```

```
102    36.103511
Length: 103, dtype: float64
```

[12]:
```
Y_max = Y.max()
Y_min = Y.min()

ax = sns.scatterplot(x=model_result.fittedvalues, y=Y)
ax.set(ylim=(Y_min, Y_max))
ax.set(xlim=(Y_min, Y_max))
ax.set_xlabel("Predicted value of Strength")
ax.set_ylabel("Observed value of Strength")

X_ref = Y_ref = np.linspace(Y_min, Y_max, 100)
plt.plot(X_ref, Y_ref, color='red', linewidth=1)
plt.show()
```



1.11 Mutliple Regression

Before embarking on any multiple regression analysis, EDA should always be performed.

Here we will:

1. Compute summary statistics
2. Plot scatterplots, 2-by-2, of all explanatory variables against the response variable.

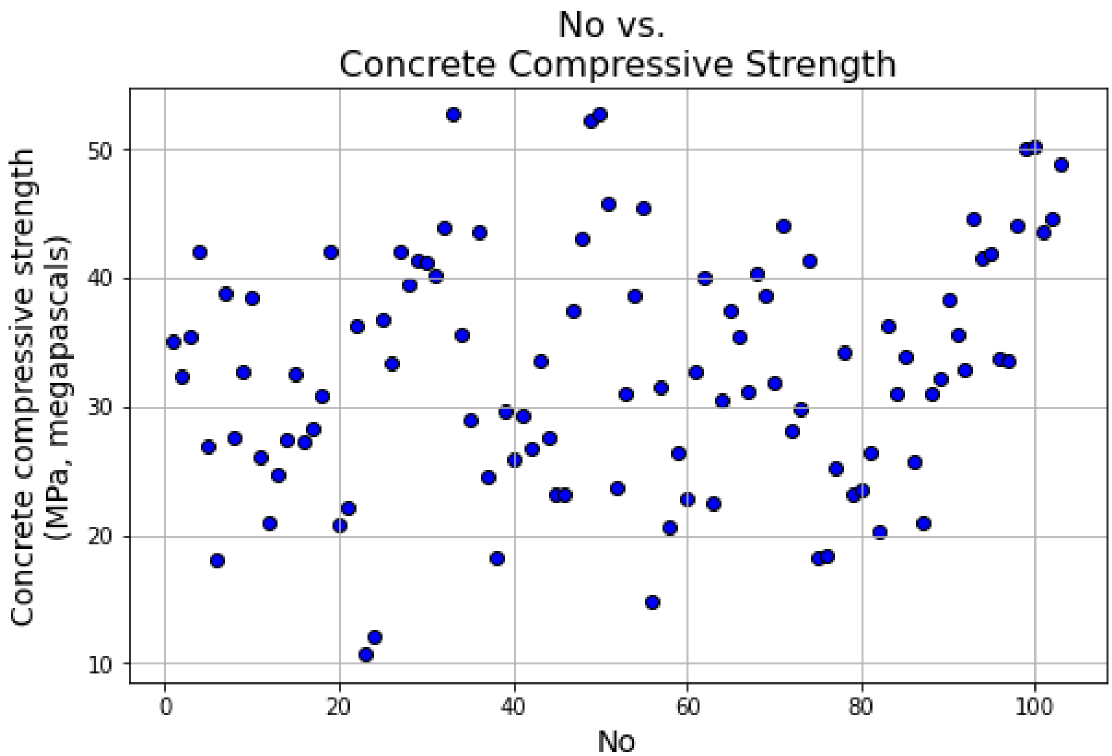
[13]:
```
con.describe()
```

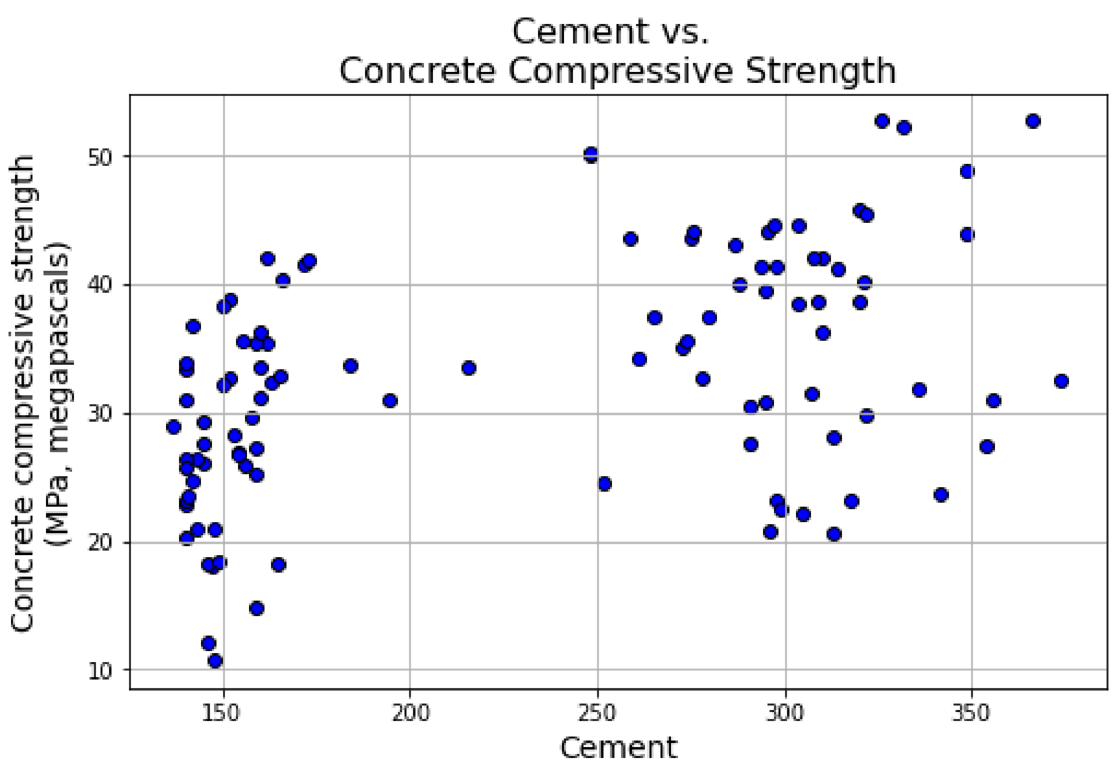
[13]:

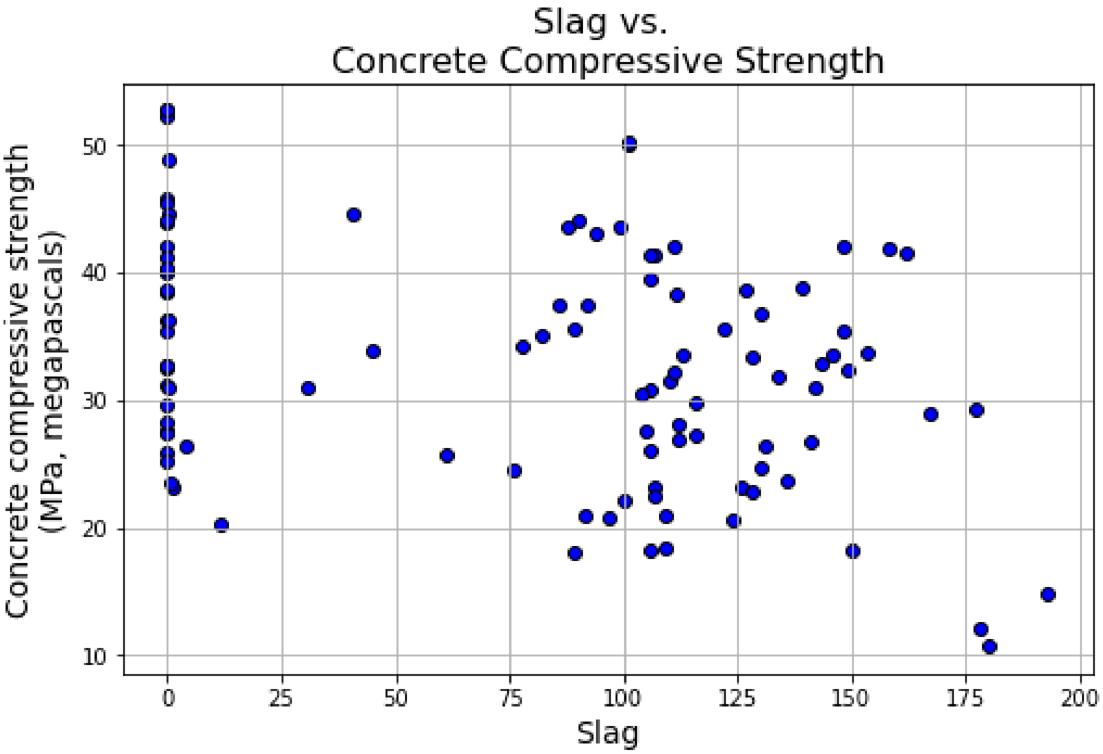
```
               No      Cement        Slag      FlyAsh       Water          SP  \
count  103.000000  103.000000  103.000000  103.000000  103.000000  103.000000
mean    52.000000  229.894175   77.973786  149.014563  197.167961    8.539806
std     29.877528   78.877230   60.461363   85.418080   20.208158    2.807530
min      1.000000  137.000000    0.000000    0.000000  160.000000    4.400000
25%     26.500000  152.000000    0.050000  115.500000  180.000000    6.000000
50%     52.000000  248.000000  100.000000  164.000000  196.000000    8.000000
75%     77.500000  303.900000  125.000000  235.950000  209.500000   10.000000
max    103.000000  374.000000  193.000000  260.000000  240.000000   19.000000

         CoarseAgg     FineAgg    Strength
count   103.000000  103.000000  103.000000
mean    883.978641  739.604854   32.840184
std      88.391393   63.342117    9.258437
min     708.000000  640.600000   10.685000
25%     819.500000  684.500000   26.220000
50%     879.000000  742.700000   32.710000
75%     952.800000  788.000000   40.065000
max    1049.900000  902.000000   52.650000
```

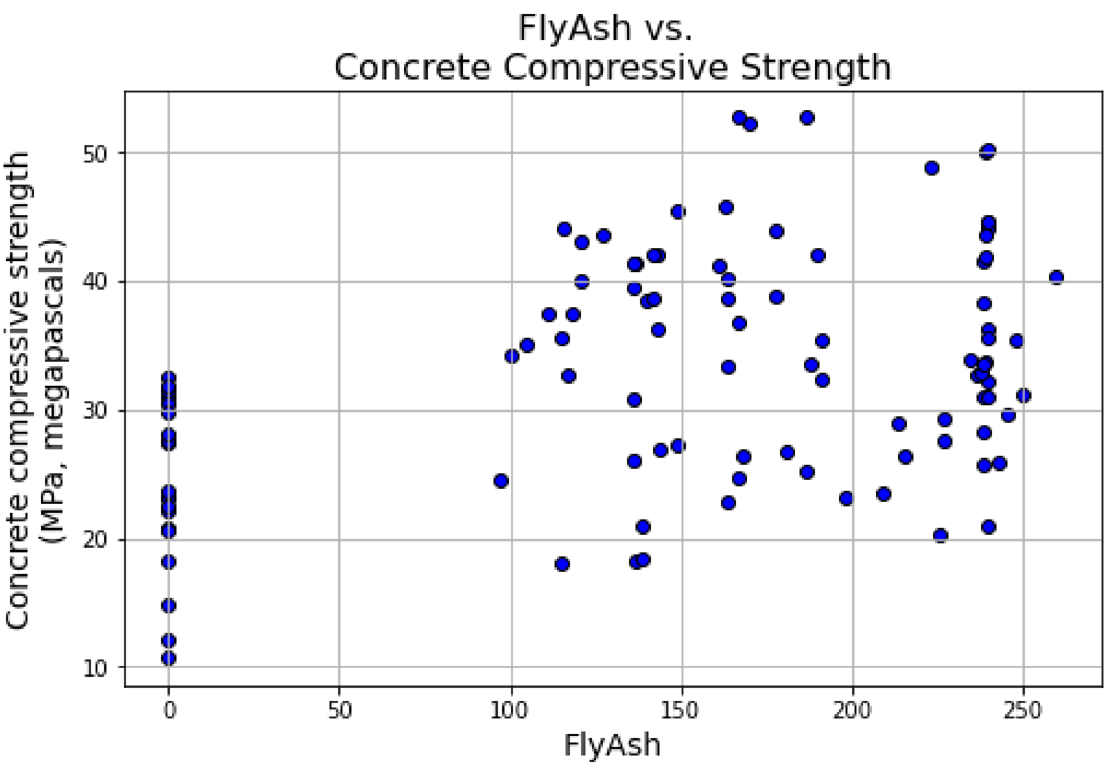
[14]:

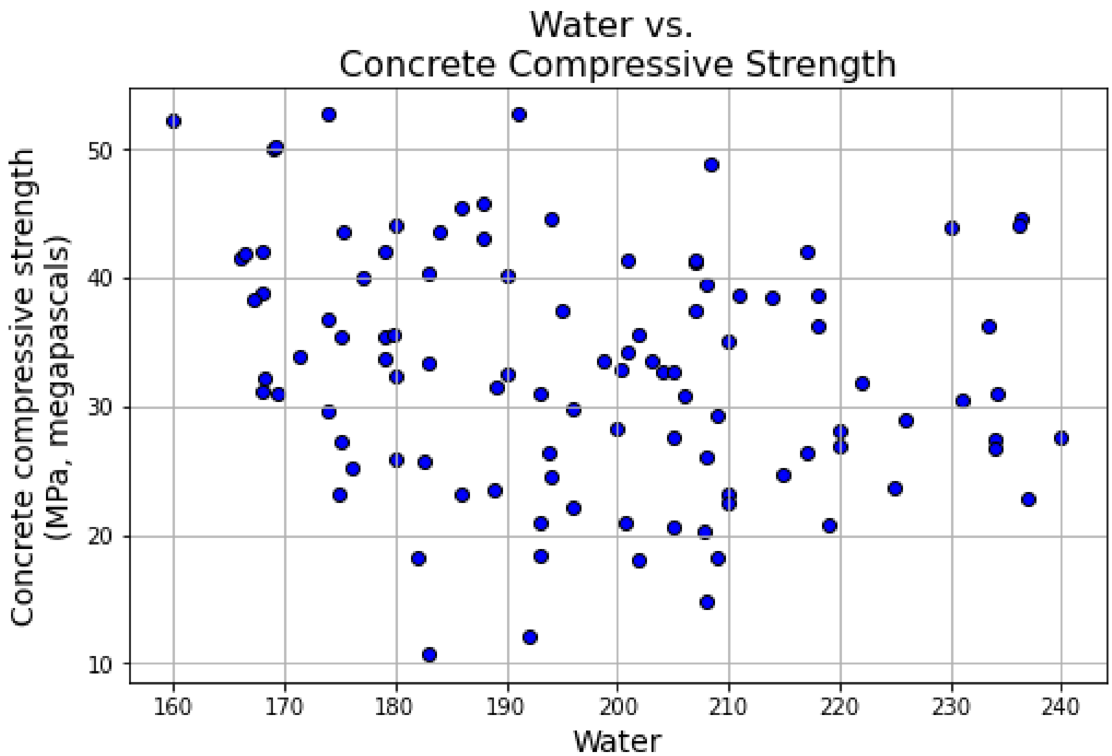
```
for c in con.columns[:-1]:
    plt.figure(figsize=(8,5))
    plt.title("{} vs. \nConcrete Compressive Strength".format(c),fontsize=16)
    plt.scatter(x=con[c],y=con['Strength'],color='blue',edgecolor='k')
    plt.grid(True)
    plt.xlabel(c,fontsize=14)
    plt.ylabel('Concrete compressive strength\n(MPa, megapascals)',fontsize=14)
    plt.show()
```

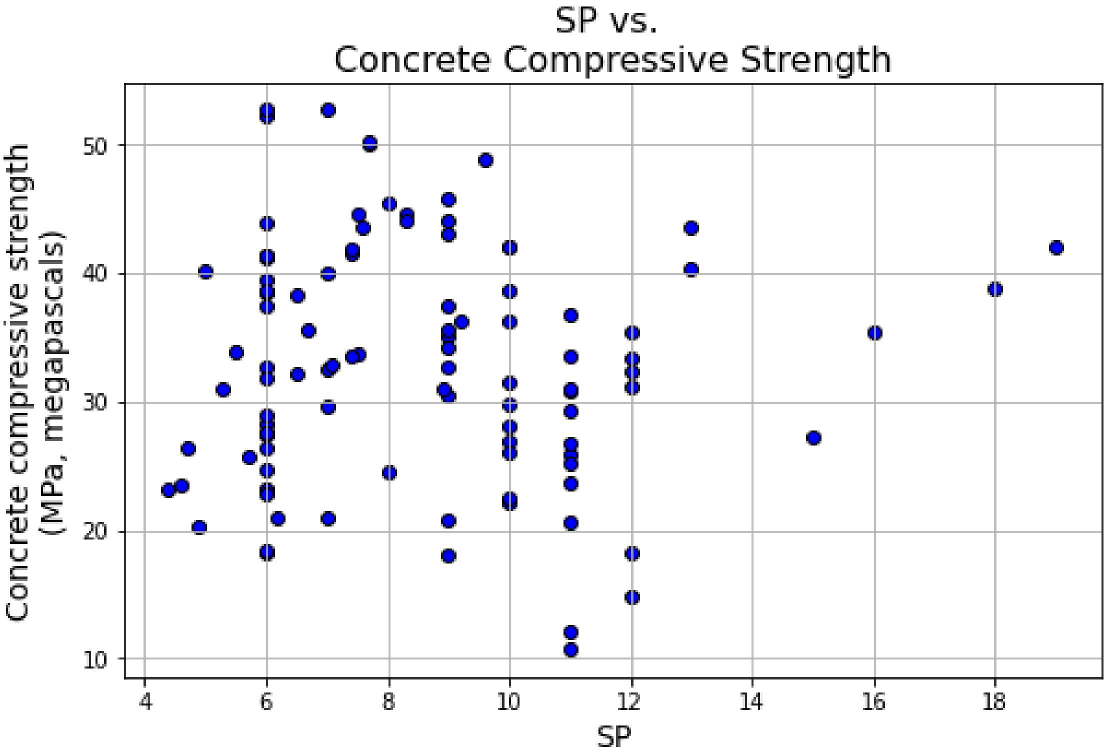

No vs.
Concrete Compressive Strength
Concrete compressive strength
(MPa, megapascals)
10
20
30
40
50
0
20
40
60
80
100
No

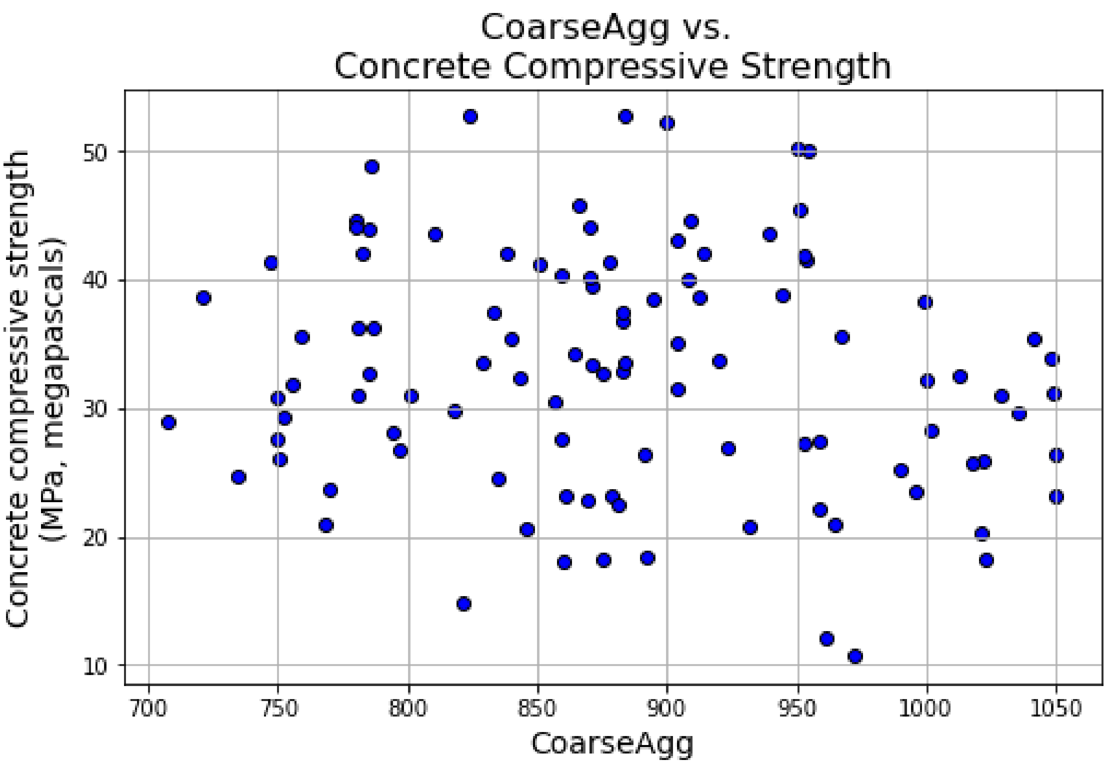

Cement vs.
Concrete Compressive Strength
Concrete compressive strength
(MPa, megapascals)
10
20
30
40
50
150
200
250
300
350
Cement

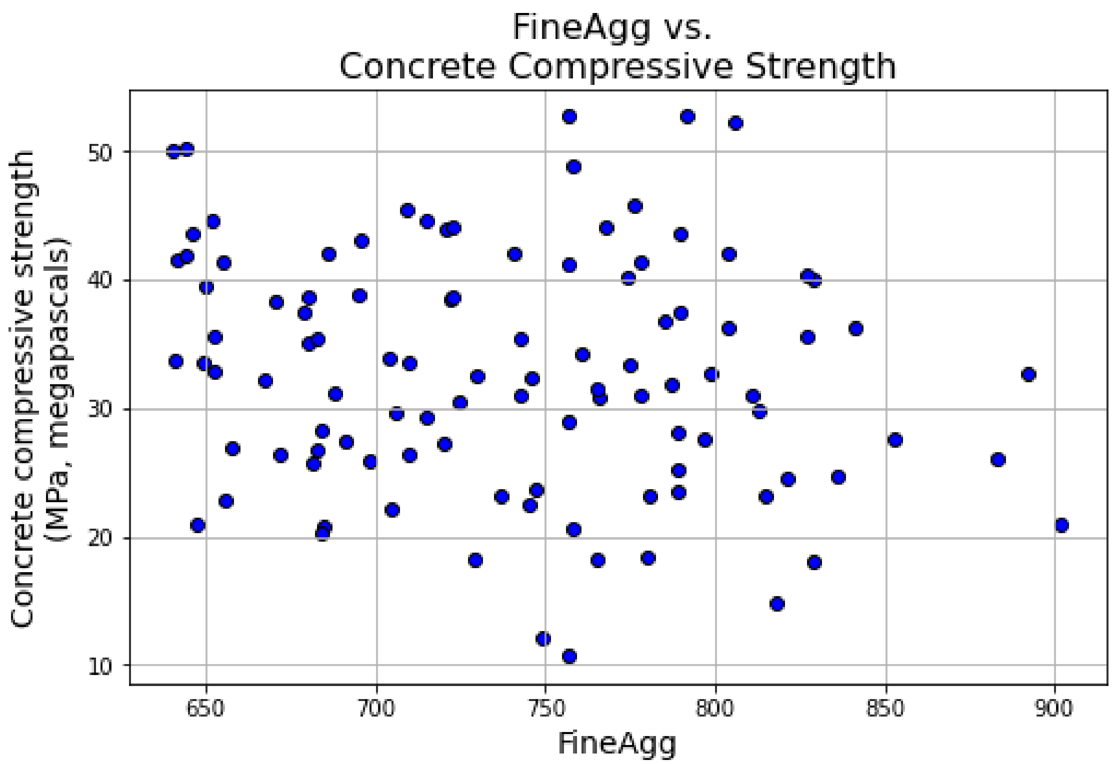

Slag vs.
Concrete Compressive Strength
Concrete compressive strength
(MPa, megapascals)
Slag

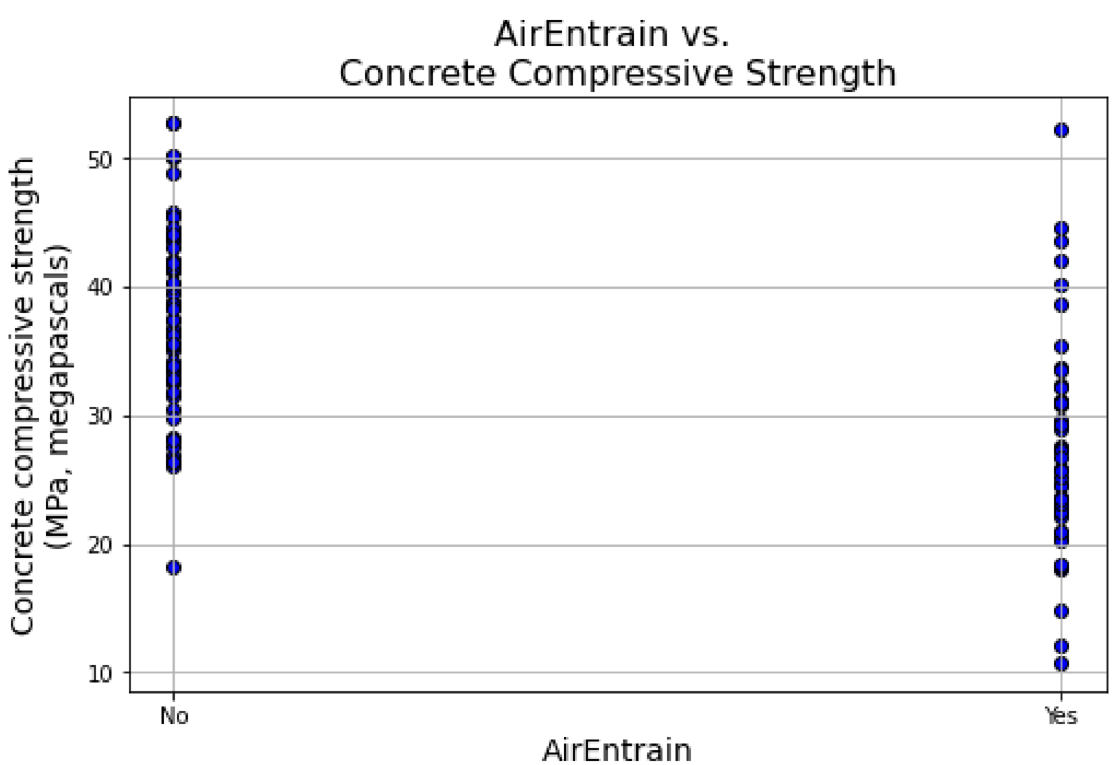

FlyAsh vs.
Concrete Compressive Strength
Concrete compressive strength
(MPa, megapascals)
FlyAsh


Water vs.
Concrete Compressive Strength
Concrete compressive strength
(MPa, megapascals)
Water
10
20
30
40
50
160
170
180
190
200
210
220
230
240


SP vs.
Concrete Compressive Strength
Concrete compressive strength
(MPa, megapascals)
SP
10
20
30
40
50
4
6
8
10
12
14
16
18


CoarseAgg vs.
Concrete Compressive Strength
Concrete compressive strength
(MPa, megapascals)
CoarseAgg
10
20
30
40
50
700
750
800
850
900
950
1000
1050


FineAgg vs.
Concrete Compressive Strength
Concrete compressive strength
(MPa, megapascals)
10
20
30
40
50
650
700
750
800
850
900
FineAgg


AirEntrain vs.
Concrete Compressive Strength
Concrete compressive strength
(MPa, megapascals)
10
20
30
40
50
No
Yes
AirEntrain

1.12 Data Preparation for Regression

Once again, define two separate data frames: 1. Y to hold the response variable (the single column "Strength") 2. X to hold all the explanatory variables

Note that we have excluded "AirEntrain" at this point because it is categorical. Including categorical variables in a linear regression requires some additional work.

[15]:
```
Y = con['Strength']
X = con[['No',
 'Cement',
 'Slag',
 'FlyAsh',
 'Water',
 'SP',
 'CoarseAgg',
 'FineAgg']]
X = sm.add_constant(X)
X.head()
```

[15]:

	const	No	Cement	Slag	FlyAsh	Water	SP	CoarseAgg	FineAgg
0	1.0	1	273.0	82.0	105.0	210.0	9.0	904.0	680.0
1	1.0	2	163.0	149.0	191.0	180.0	12.0	843.0	746.0
2	1.0	3	162.0	148.0	191.0	179.0	16.0	840.0	743.0
3	1.0	4	162.0	148.0	190.0	179.0	19.0	838.0	741.0
4	1.0	5	154.0	112.0	144.0	220.0	10.0	923.0	658.0

1.13 Kitchen sink model

Usual practice is to start with a "kitchen sink" model, which includes **all** the (numerical) explanatory variables.

The Statsmodels OLS output gives us some warnings at the bottom of the output. These concern collinearity, due to nuisance variables that should eventually be eliminated, or shrunken-see below. We can ignore these at this early stage of the modeling process.

[16]:
```
ks = sm.OLS(Y, X)
ks_res =ks.fit()
ks_res.summary()
```

[16]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:               Strength   R-squared:                       0.827
Model:                            OLS   Adj. R-squared:                  0.812
```

```
Method:                 Least Squares   F-statistic:                     56.21
Date:                Sun, 10 Sep 2023   Prob (F-statistic):           1.68e-32
Time:                        12:21:46   Log-Likelihood:                -284.49
No. Observations:                 103   AIC:                             587.0
Df Residuals:                      94   BIC:                             610.7
Df Model:                           8
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        115.2834    142.786      0.807      0.421    -168.222     398.789
No            -0.0077      0.021     -0.372      0.711      -0.049       0.033
Cement         0.0826      0.047      1.758      0.082      -0.011       0.176
Slag          -0.0225      0.065     -0.346      0.730      -0.152       0.107
FlyAsh         0.0668      0.048      1.380      0.171      -0.029       0.163
Water         -0.2165      0.142     -1.520      0.132      -0.499       0.066
SP             0.2518      0.213      1.181      0.241      -0.172       0.675
CoarseAgg     -0.0479      0.056     -0.857      0.393      -0.159       0.063
FineAgg       -0.0356      0.057     -0.622      0.536      -0.149       0.078
==============================================================================
Omnibus:                        2.168   Durbin-Watson:                   1.715
Prob(Omnibus):                  0.338   Jarque-Bera (JB):                2.183
Skew:                          -0.309   Prob(JB):                        0.336
Kurtosis:                       2.644   Cond. No.                     4.36e+05
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 4.36e+05. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

1.14 Categorical explanatory variables

To deal with categorical explanatory variables in **R**, we need to convert the variable to a *factor* data type and let R construct the n-1 dummy columns behind the scenes.

In Python, we can use either the manual approach (create a matrix of dummy variables ourselves) or the automatic approach (let the algorithm sort it out behind the scenes). We recommend the manual approach because dealing intelligently with categorical variables in real-world data *almost always* involves significant work.

Specifically: we typically need to change the granularity of the variable to provide more generalizable results.

1.14.1 Create a matrix of dummy variables

Many different libraries in Python provide many different routines for encoding categorical variables. All of these routines bypass the drudgery of writing IF statements to map from categorical values to (0, 1) values. Here we will use Pandas's aptly-named `get_dummies()` method.

In this approach, we pass `get_dummies()` a column in a data frame and it creates a full matrix of zero-one values–this is also knwon as **one-hot encoding**. In other words, it gives us a matrix with 103 rows (because we have 103 rows in the "Concrete Strength" data set and two columns (because the "AirEntrain" variable has two values: Yes and No).

[17]:
```
AirEntrain_d = pd.get_dummies(con['AirEntrain'])
AirEntrain_d
```

[17]:
```
     No  Yes
0     1    0
1     0    1
2     0    1
3     1    0
4     1    0
..   ..  ...
98    1    0
99    1    0
100   0    1
101   0    1
102   1    0

[103 rows x 2 columns]
```

The "Yes" and "No" column headings can be problematic, especially if we have to convert many categorical variables with Yes/No values. Accordingly, we need to make some changes to the default dummy matrix:

1. We use AirEntrain=No as the baseline for the dummy variable. As such, we need to drop the "No" column from the matrix before passing it to the regression.
2. We can embed the choice of baseline into the the dummy column names. This makes it easier to interpret the regression coefficients

[18]:
```
AirEntrain_d.drop(columns='No', inplace=True)
AirEntrain_d.rename(columns={'Yes': 'AirEntrain_Yes'}, inplace=True)
AirEntrain_d.head(3)
```

[18]:
```
   AirEntrain_Yes
0               0
1               1
2               1
```

1.14.2 Adding the dummy columns to the existing X matrix

[19]:
```
fullX = pd.concat([X, AirEntrain_d['AirEntrain_Yes']], axis=1)
fullX.head()
```

[19]:
```
   const  No  Cement   Slag  FlyAsh  Water    SP  CoarseAgg  FineAgg  \
0    1.0   1   273.0   82.0   105.0  210.0   9.0      904.0    680.0
1    1.0   2   163.0  149.0   191.0  180.0  12.0      843.0    746.0
2    1.0   3   162.0  148.0   191.0  179.0  16.0      840.0    743.0
3    1.0   4   162.0  148.0   190.0  179.0  19.0      838.0    741.0
4    1.0   5   154.0  112.0   144.0  220.0  10.0      923.0    658.0

   AirEntrain_Yes
0               0
1               1
2               1
3               0
4               0
```

1.14.3 Running the full regression

We can now rerun the regression, including the categorical variable.

[20]:
```
ks2 = sm.OLS(Y, fullX)
ks2_res = ks2.fit()
ks2_res.summary()
```

[20]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:               Strength   R-squared:                       0.924
Model:                            OLS   Adj. R-squared:                  0.916
Method:                 Least Squares   F-statistic:                     125.1
Date:                Sun, 10 Sep 2023   Prob (F-statistic):           5.83e-48
Time:                        12:21:46   Log-Likelihood:                -242.38
No. Observations:                 103   AIC:                             504.8
Df Residuals:                      93   BIC:                             531.1
Df Model:                           9
Covariance Type:            nonrobust
================================================================================
==
                     coef    std err          t      P>|t|      [0.025
0.975]
--------------------------------------------------------------------------------
--
const             41.5005     95.617      0.434      0.665    -148.375
231.376
```

```
No                 -0.0173      0.014     -1.251      0.214      -0.045
0.010
Cement              0.0962      0.031      3.063      0.003       0.034
0.159
Slag                0.0157      0.044      0.359      0.720      -0.071
0.102
FlyAsh              0.0869      0.032      2.684      0.009       0.023
0.151
Water              -0.1380      0.095     -1.446      0.151      -0.328
0.051
SP                  0.1902      0.143      1.334      0.186      -0.093
0.473
CoarseAgg          -0.0160      0.037     -0.428      0.669      -0.090
0.058
FineAgg            -0.0021      0.038     -0.053      0.957      -0.078
0.074
AirEntrain_Yes     -6.0683      0.559    -10.848      0.000      -7.179
-4.957
==============================================================================
Omnibus:                        4.217   Durbin-Watson:                   1.637
Prob(Omnibus):                  0.121   Jarque-Bera (JB):                3.635
Skew:                           0.351   Prob(JB):                        0.162
Kurtosis:                       3.594   Cond. No.                     4.37e+05
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 4.37e+05. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

We observe that:

1. The R-squared value has improved considerably and is now equal to 0.924.
2. There is still a problem with collinearity.

1.15 Using R-like formulas

As mentioned previously, R was used in statistics long before Python was popular. As a consequence, some of the data science libraries for Python mimic the R way of doing things. This makes it much easier for people who know R to transition to Python. If, however, you do not know R, it can add some confusion.

Having said this, formula notation in R turns out to be very useful. Instead of defining separate Y and X matrices, we simply pass R a formula of the form "Y ~ X1, X2, … Xn" and it takes care of the rest. It turns out that Statsmodels includes a whole library for doing things the R way. Two things to know:

1. You have to import the statsmodels.formula.api library instead of (or, more typically, in

addition to) the statsmodels.api library
2. The method names in the "formula" api are lowercase (e.g., `ols()` instead of `OLS()`

Let us do this now, in "R" style.

[21]:
```
import statsmodels.formula.api as smf
ksf =  smf.ols(' Strength ~ No + Cement + Slag + Water + CoarseAgg + FlyAsh +␣
 ↪SP + FineAgg + AirEntrain', data=con)
ksf_res = ksf.fit()
ksf_res.summary()
```

[21]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:               Strength   R-squared:                       0.924
Model:                            OLS   Adj. R-squared:                  0.916
Method:                 Least Squares   F-statistic:                     125.1
Date:                Sun, 10 Sep 2023   Prob (F-statistic):           5.83e-48
Time:                        12:21:46   Log-Likelihood:                -242.38
No. Observations:                 103   AIC:                             504.8
Df Residuals:                      93   BIC:                             531.1
Df Model:                           9
Covariance Type:            nonrobust
================================================================================
=====
                        coef    std err          t      P>|t|      [0.025
0.975]
--------------------------------------------------------------------------------
-----
Intercept            41.5005     95.617      0.434      0.665    -148.375
231.376
AirEntrain[T.Yes]    -6.0683      0.559    -10.848      0.000      -7.179
-4.957
No                   -0.0173      0.014     -1.251      0.214      -0.045
0.010
Cement                0.0962      0.031      3.063      0.003       0.034
0.159
Slag                  0.0157      0.044      0.359      0.720      -0.071
0.102
Water                -0.1380      0.095     -1.446      0.151      -0.328
0.051
CoarseAgg            -0.0160      0.037     -0.428      0.669      -0.090
0.058
FlyAsh                0.0869      0.032      2.684      0.009       0.023
0.151
SP                    0.1902      0.143      1.334      0.186      -0.093
0.473
```

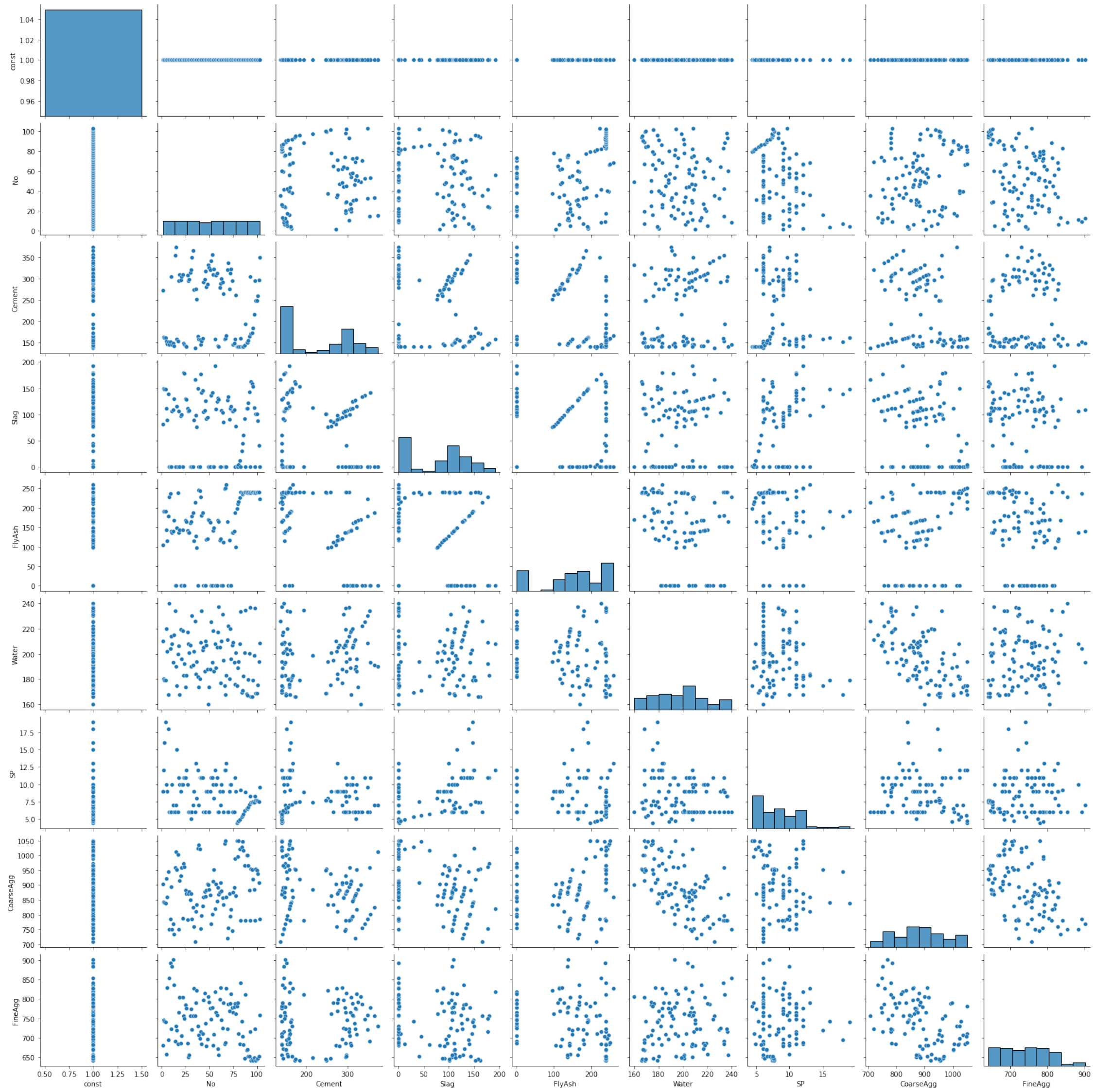
```
FineAgg               -0.0021      0.038     -0.053      0.957      -0.078
0.074
==============================================================================
Omnibus:                        4.217   Durbin-Watson:                   1.637
Prob(Omnibus):                  0.121   Jarque-Bera (JB):                3.635
Skew:                           0.351   Prob(JB):                        0.162
Kurtosis:                       3.594   Cond. No.                     4.37e+05
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 4.37e+05. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

1.16 Checking for colinearity

1.16.1 Scatterplot matrix

We can plot scatterplot matrix on our original X matrix using Seaborn's handy `pairplot()` method. A nice feature of this presentation is a histogram for each variable. **Note** that this may take a few seconds to generate so you have to be patient.

[22]:
```
import seaborn as sns
sns.pairplot(X);
```



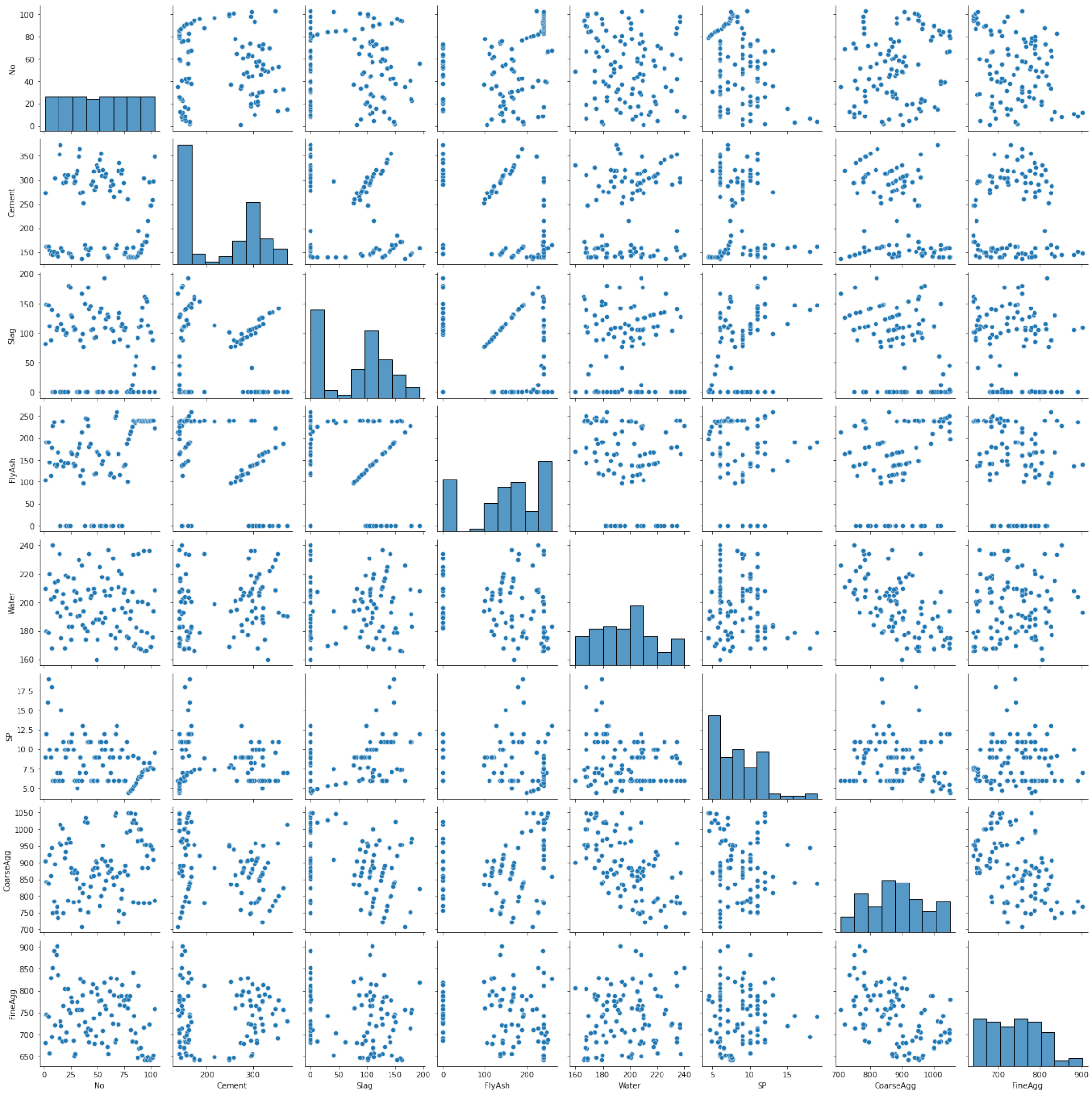
1.16.2 Restricting variables in the scatterplot matrix

With wide data sets (having many columns), the scatterplots become unreadable. Thus, it is often better to restrict the variables in the scatterplot matrix to a named set in order to maximize readability. Here we exclud the constant, response variable, and all dummy columns.

A few things that catch the eye in the scatterplot matrix:

1. The "No" variable (experiment number) does not appear to be correlated with any other variable. That is good news—we should not expect it to in a well-run experiment.
2. There is some linearity and other strangeness in the relationships between "FlyAsh", "Slag", and "Cement". This suggests problems with the experimental design. Unfortunately, these problems cannot be fixed in the data analysis stage.

[23]:
```
sns.pairplot(X[['No',
 'Cement',
 'Slag',
 'FlyAsh',
 'Water',
 'SP',
 'CoarseAgg',
 'FineAgg']]);
```



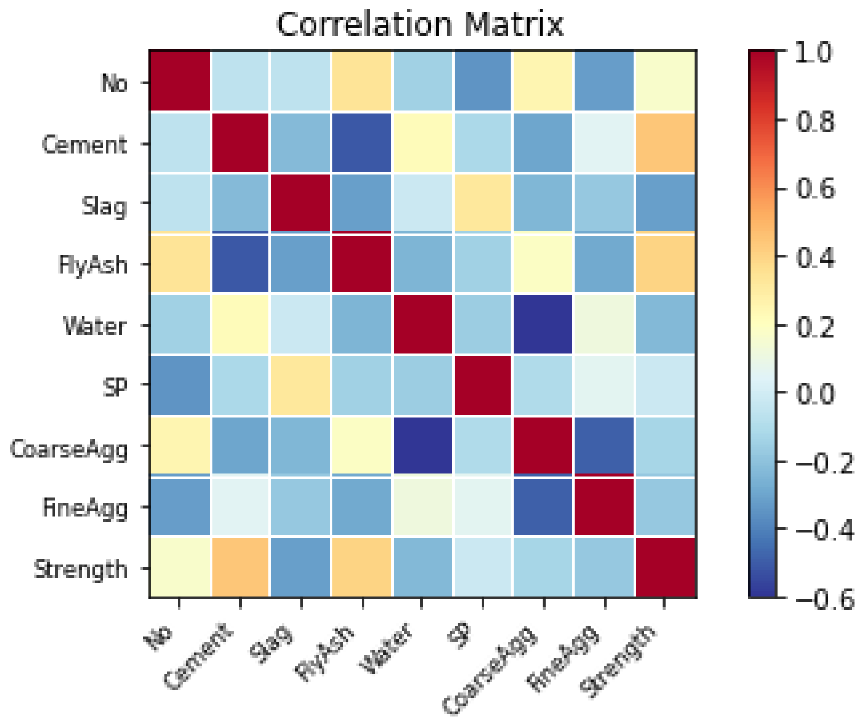
1.16.3 Correlation matrix

If the scatterplot matrix remains too hard to read, one can always revert to a simple correlation matrix.

[24]:
```
round(con.corr(),2)
```

[24]:
```
              No  Cement  Slag  FlyAsh  Water    SP  CoarseAgg  FineAgg  \
No          1.00   -0.03 -0.08    0.34  -0.14 -0.33       0.22    -0.31
Cement     -0.03    1.00 -0.24   -0.49   0.22 -0.11      -0.31     0.06
Slag       -0.08   -0.24  1.00   -0.32  -0.03  0.31      -0.22    -0.18
FlyAsh      0.34   -0.49 -0.32    1.00  -0.24 -0.14       0.17    -0.28
Water      -0.14    0.22 -0.03   -0.24   1.00 -0.16      -0.60     0.11
SP         -0.33   -0.11  0.31   -0.14  -0.16  1.00      -0.10     0.06
CoarseAgg   0.22   -0.31 -0.22    0.17  -0.60 -0.10       1.00    -0.49
FineAgg    -0.31    0.06 -0.18   -0.28   0.11  0.06      -0.49     1.00
Strength    0.19    0.46 -0.33    0.41  -0.22 -0.02      -0.15    -0.17

           Strength
No             0.19
Cement         0.46
Slag          -0.33
FlyAsh         0.41
Water         -0.22
SP            -0.02
CoarseAgg     -0.15
FineAgg       -0.17
Strength       1.00
```

[25]:
```
corr = con[:-1].corr()
corr
from statsmodels.graphics.correlation import plot_corr
fig = plot_corr(corr,xnames=corr.columns)
```



1.17 Model Refinement and Feature Selection

The kitchen sink model is unlikely to be the best model. At the very least, we need to remove variables that should not be in the model for **methodological** reasons, such as collinearity. Then, depending on our philosophical view on such things, we can go into data mining mode and attempt to generate the "best" model by removing or adding explanatory variables. Two clarifications:

1. The **best** model is typically defined in terms of the trade-off between goodness of fit (e.g., R^2) and model complexity (the number of explanatory variables). This trade-off provides the rationale for the *adjusted* R^2 measure. Given two models with similar explanatory power, the one with the fewest explanatory variables is deemed better.

2. **Data mining mode** means we suspend our knowledge about the underlying domain and instead focus on technical measures of explanatory power. In this mode, we keep our theories about cause and effect to ourselves: If the measure indicates a variable has explanatory power, we leave it in the model; if the measure indicates the variable has low explanatory power, we take it out of the model. Many different heuristic measures of explanatory power exist, including the p-value of the coefficient and the more sophistical measures (AIC, Mallows Cp) used by R.

These will be left to a later example.

1.17.1 Manual stepwise refinement

When we do manual stepwise refinement, the heuristic is to start with the kitchen sink model and remove the variable with the highest p-value (probability of zero slope).

If we scroll up to the results of the kitchen sink model, we see that the variable with the highest p-value is "FineAgg". If we are using the matrix version of the `OLS()` method, we can drop the column from the X matrix.

[26]:
```
X1 = fullX.drop(columns='FineAgg', inplace=False)
mod1 = sm.OLS(Y, X1)
mod1_res = mod1.fit()
mod1_res.summary()
```

[26]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:               Strength   R-squared:                       0.924
Model:                            OLS   Adj. R-squared:                  0.917
Method:                 Least Squares   F-statistic:                     142.2
Date:                Sun, 10 Sep 2023   Prob (F-statistic):           4.73e-49
Time:                        12:21:52   Log-Likelihood:                -242.38
No. Observations:                 103   AIC:                             502.8
Df Residuals:                      94   BIC:                             526.5
Df Model:                           8
Covariance Type:            nonrobust
================================================================================
==
                     coef    std err          t      P>|t|      [0.025
0.975]
--------------------------------------------------------------------------------
--
const             36.4097      8.674      4.197      0.000      19.186
53.633
No                -0.0178      0.011     -1.674      0.097      -0.039
0.003
Cement             0.0978      0.005     18.070      0.000       0.087
0.109
Slag               0.0180      0.006      2.819      0.006       0.005
0.031
FlyAsh             0.0887      0.005     17.367      0.000       0.079
0.099
Water             -0.1330      0.019     -7.131      0.000      -0.170
-0.096
SP                 0.1950      0.109      1.791      0.077      -0.021
0.411
CoarseAgg         -0.0141      0.005     -2.964      0.004      -0.023
```

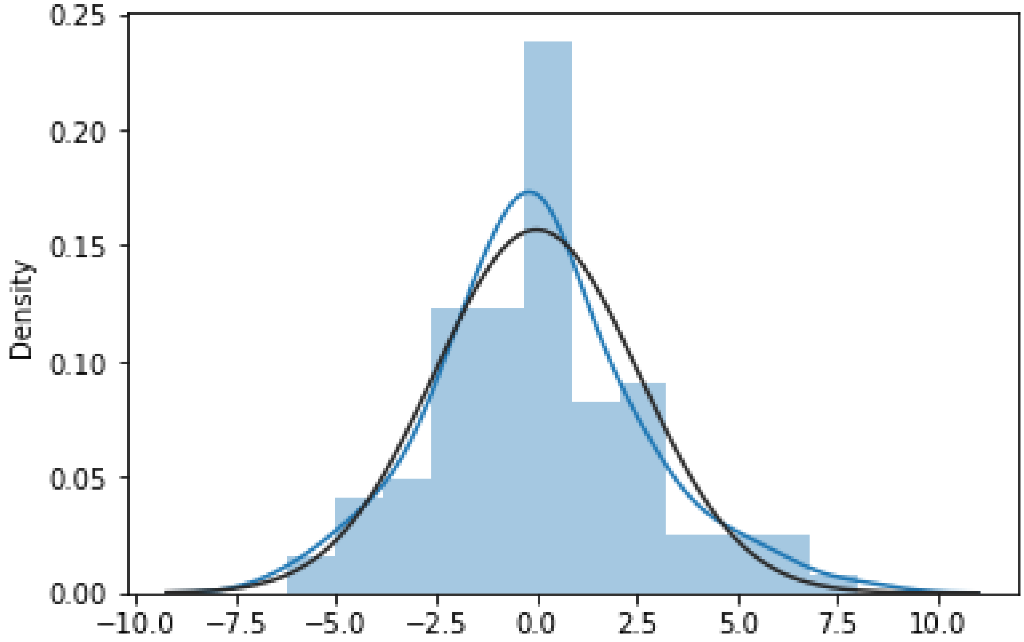
```
-0.005
AirEntrain_Yes    -6.0707      0.555    -10.946      0.000      -7.172
-4.970
==============================================================================
Omnibus:                        4.255   Durbin-Watson:                   1.637
Prob(Omnibus):                  0.119   Jarque-Bera (JB):                3.680
Skew:                           0.352   Prob(JB):                        0.159
Kurtosis:                       3.601   Cond. No.                     3.15e+04
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 3.15e+04. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

1.18 Regression diagnostics

As we did above for simple regression, we can generate diagnostic plots to determine whether the resulting regression model is valid.

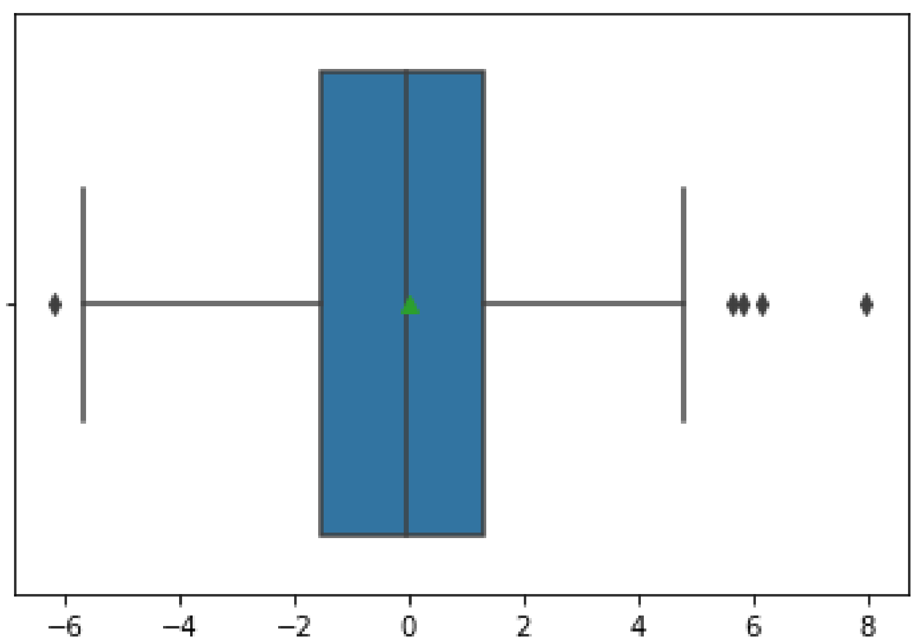
[27]:
```
from scipy import stats
sns.distplot(mod1_res.resid, fit=stats.norm);
```

```
/Users/markasch/opt/anaconda3/lib/python3.9/site-
packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a
deprecated function and will be removed in a future version. Please adapt your
code to use either `displot` (a figure-level function with similar flexibility)
or `histplot` (an axes-level function for histograms).
  warnings.warn(msg, FutureWarning)
```



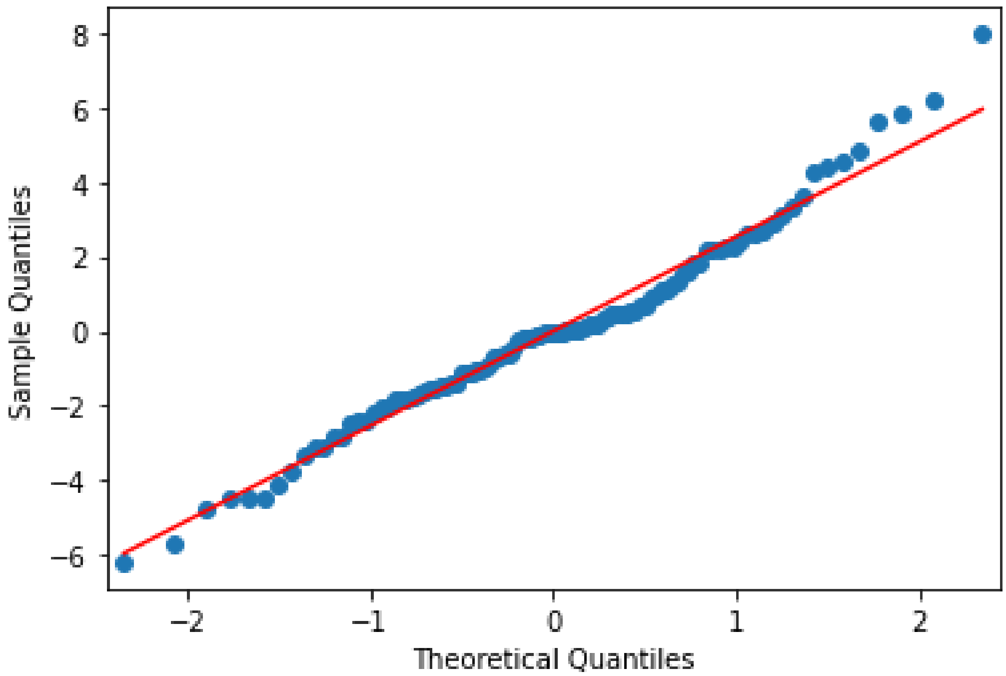
[28]:

```
sns.boxplot(x=mod1_res.resid, showmeans=True);
```



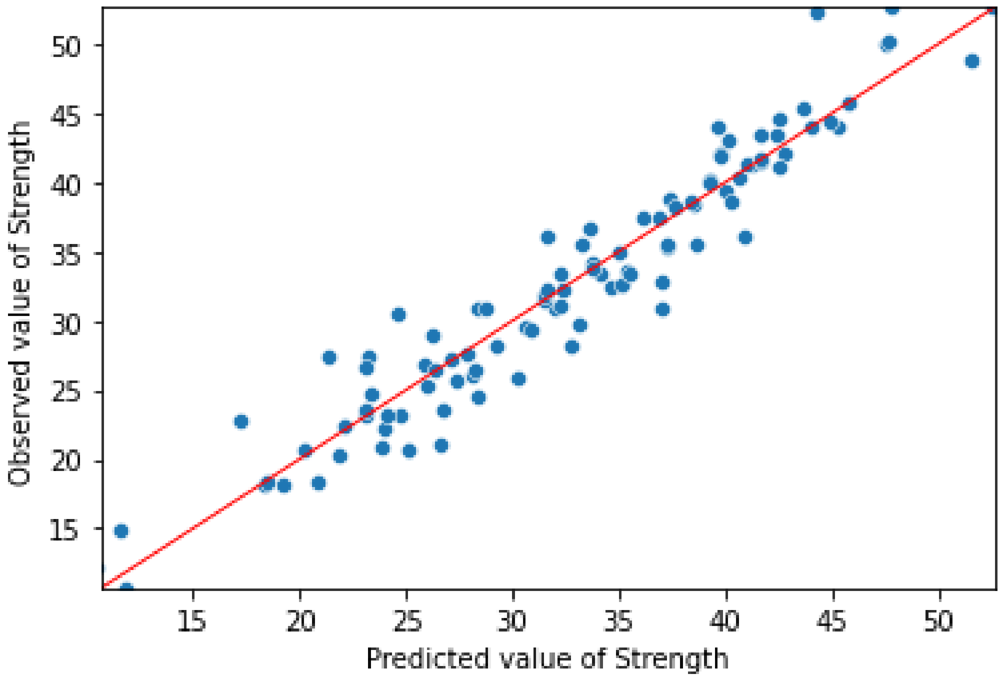
[29]:

```
sm.qqplot(mod1_res.resid, line='s');
```



We proceed, once again, to plot fitted vs. observed values.

[30]:
```
import matplotlib.pyplot as plt
import numpy as np

Y_max = Y.max()
Y_min = Y.min()

ax = sns.scatterplot(x=mod1_res.fittedvalues, y=Y)
ax.set(ylim=(Y_min, Y_max))
ax.set(xlim=(Y_min, Y_max))
ax.set_xlabel("Predicted value of Strength")
ax.set_ylabel("Observed value of Strength")

X_ref = Y_ref = np.linspace(Y_min, Y_max, 100)
plt.plot(X_ref, Y_ref, color='red', linewidth=1)
plt.show()
```

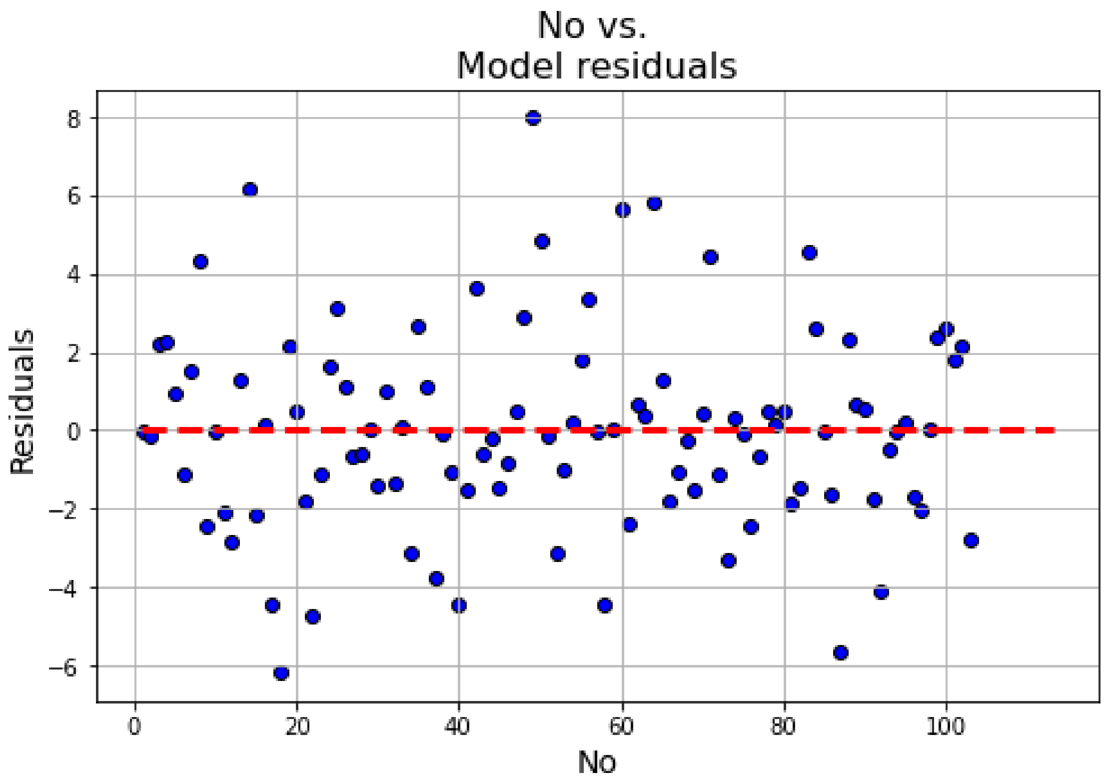


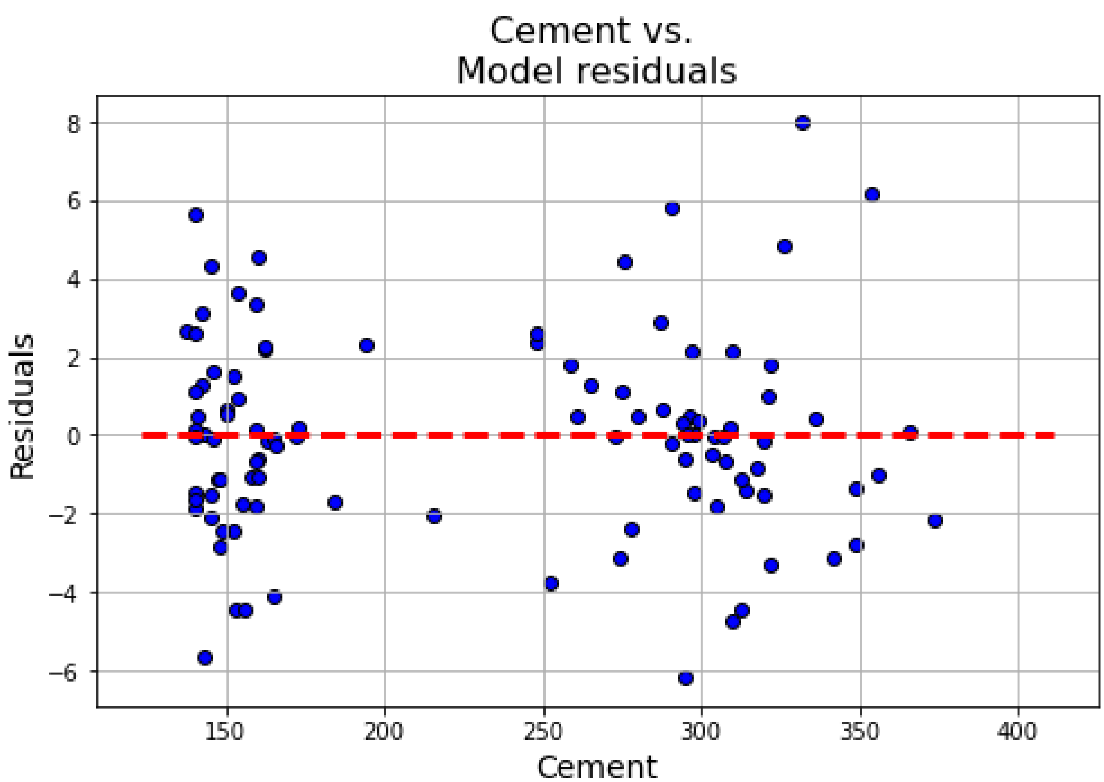
The improvement is considerable! Our model now captures, quite reliably, most of the range of compressive strength.

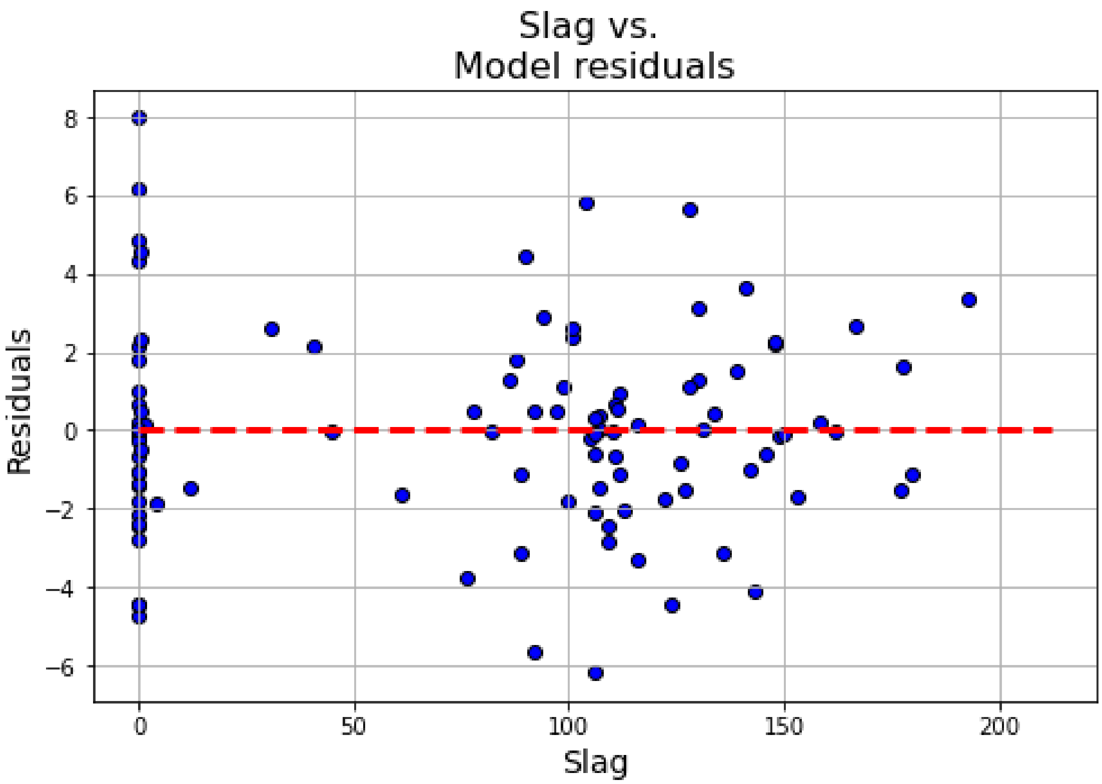
1.18.1 Alternative diagnostic plots

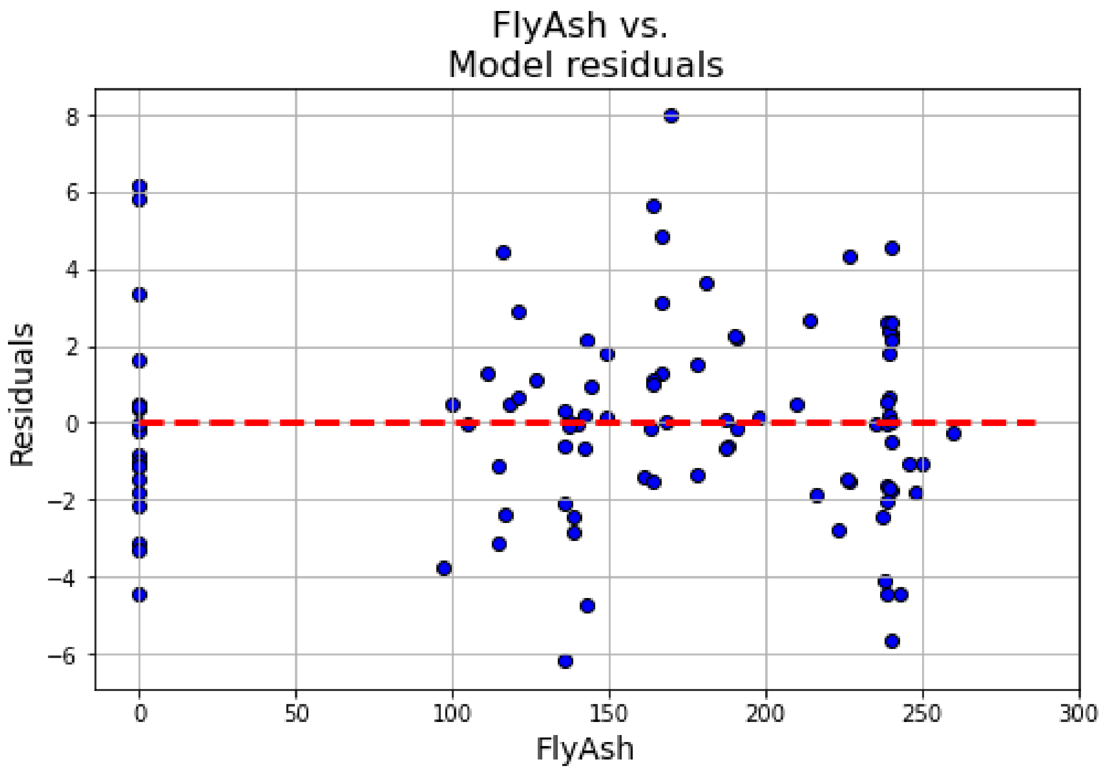
We can plot individual residuals and overall fitted resiuduals.

[31]:
```
for c in X1.columns[1:-1]:
    plt.figure(figsize=(8,5))
    plt.title("{} vs. \nModel residuals".format(c),fontsize=16)
    plt.scatter(x=X1[c],y=mod1_res.resid,color='blue',edgecolor='k')
    plt.grid(True)
    xmin=min(X1[c])
    xmax = max(X1[c])
    plt.hlines(y=0,xmin=xmin*0.9,xmax=xmax*1.1,color='red',linestyle='--',lw=3)
    plt.xlabel(c,fontsize=14)
    plt.ylabel('Residuals',fontsize=14)
    plt.show()
```

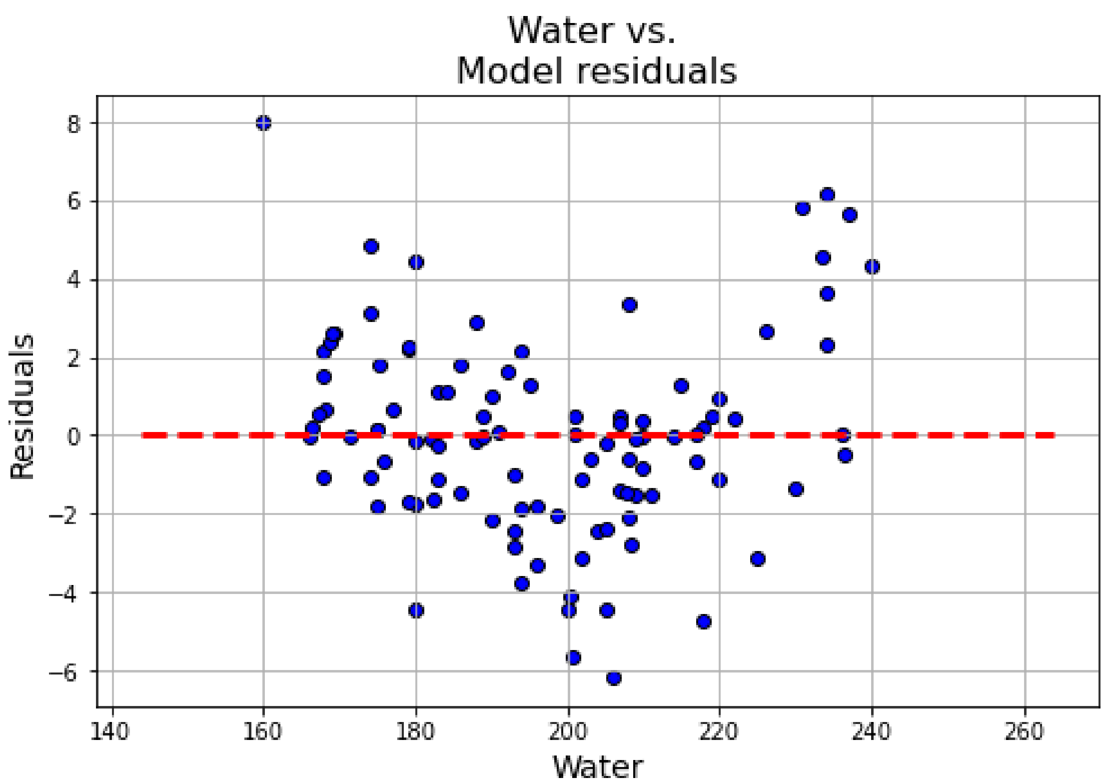

No vs.
Model residuals
Residuals
No
8
6
4
2
0
−2
−4
−6
0
20
40
60
80
100


Cement vs.
Model residuals
Residuals
Cement
8
6
4
2
0
−2
−4
−6
150
200
250
300
350
400

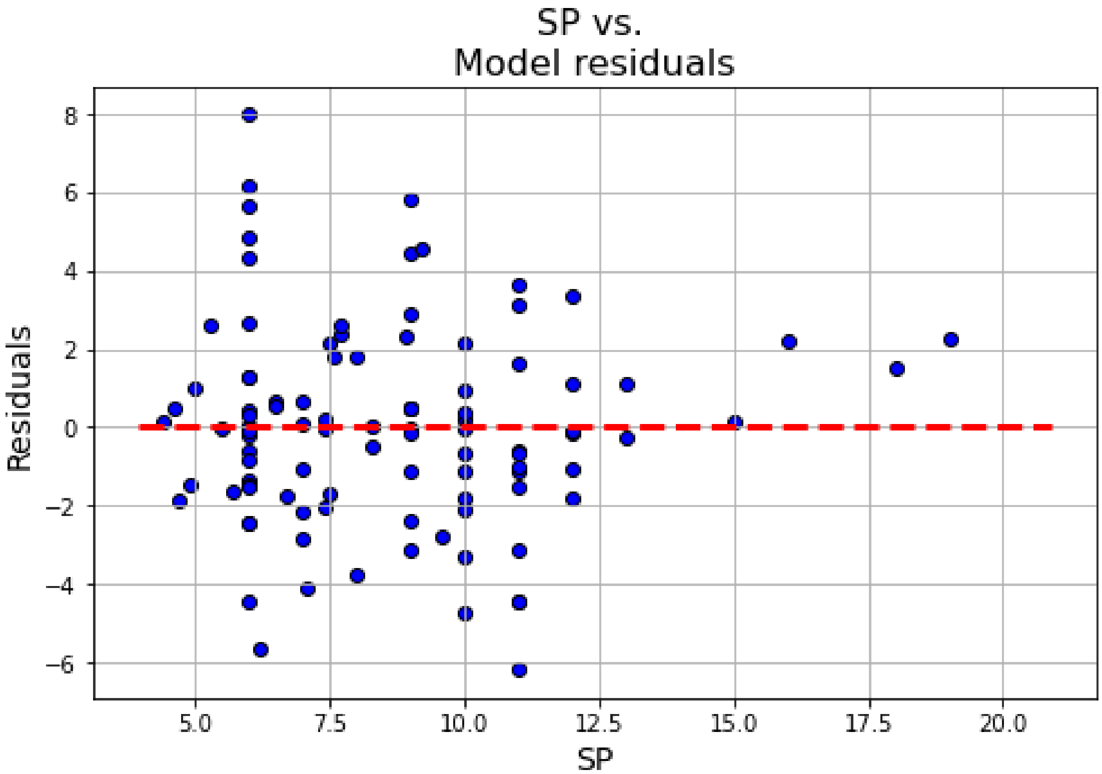

Slag vs.
Model residuals
Residuals
Slag
8
6
4
2
0
−2
−4
−6
0
50
100
150
200



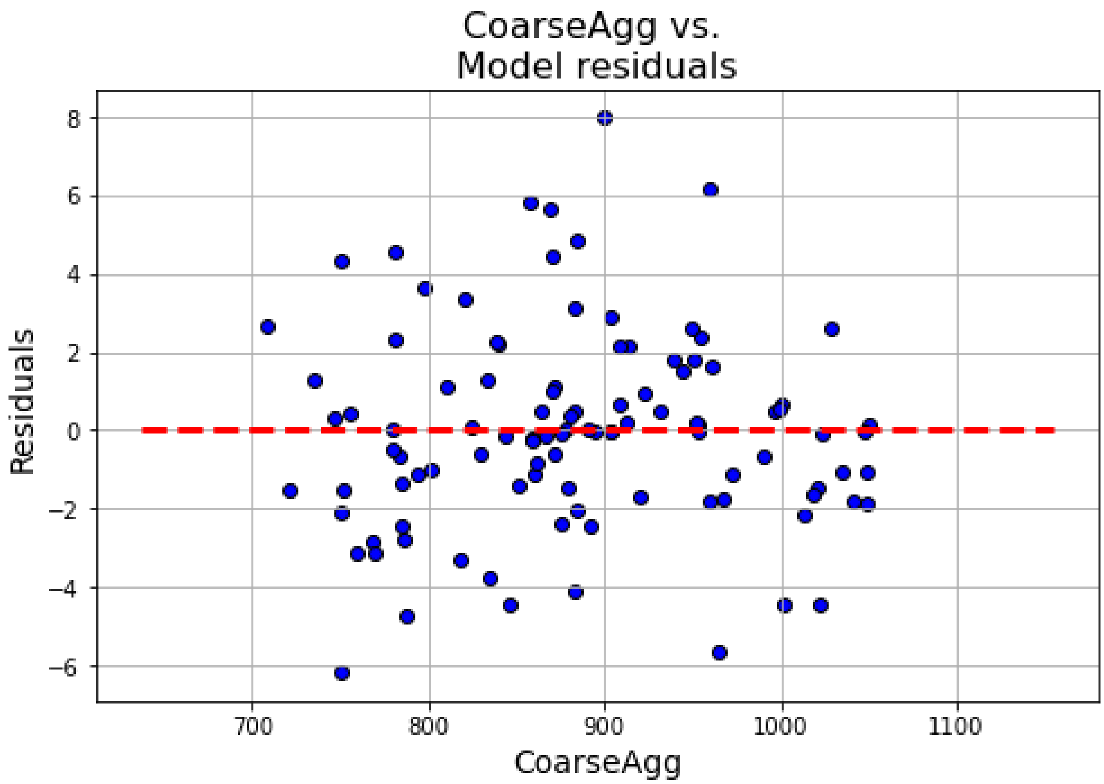
FlyAsh vs.
Model residuals
Residuals
FlyAsh
8
6
4
2
0
−2
−4
−6
0
50
100
150
200
250
300



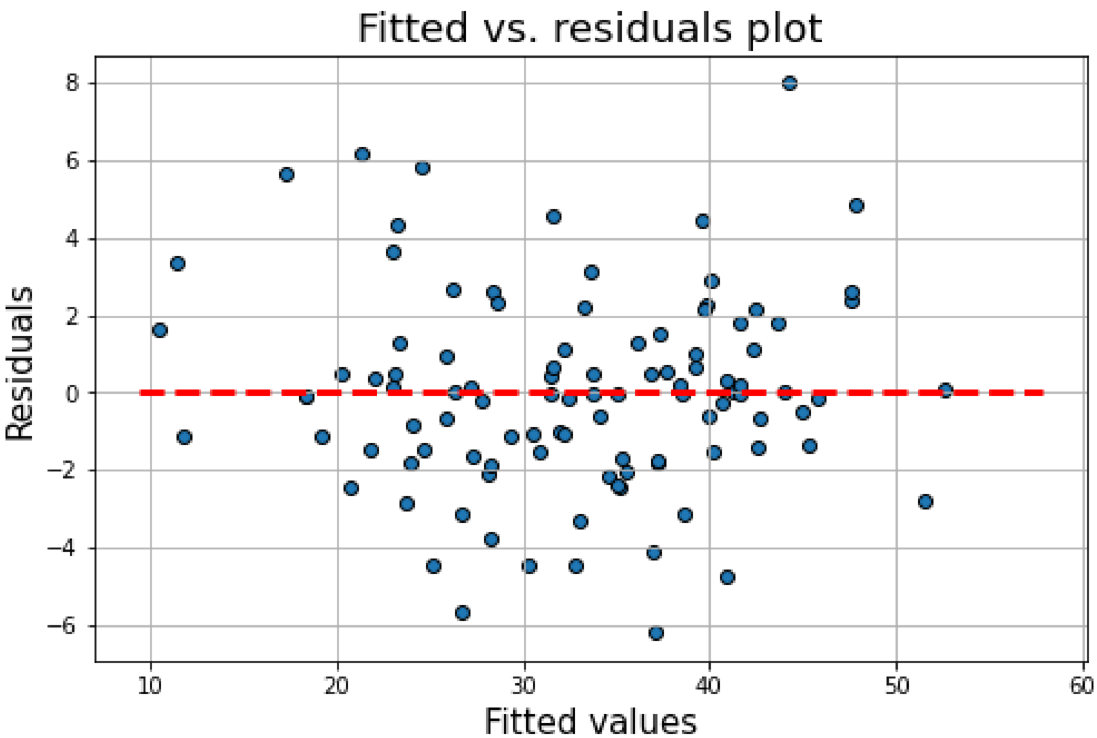
Water vs.
Model residuals
Residuals
Water
8
6
4
2
0
−2
−4
−6
140
160
180
200
220
240
260



SP vs.
Model residuals
Residuals
SP
8
6
4
2
0
−2
−4
−6
5.0
7.5
10.0
12.5
15.0
17.5
20.0



[32]:
```
plt.figure(figsize=(8,5))
p=plt.scatter(x=mod1_res.fittedvalues,y=mod1_res.resid,edgecolor='k')
xmin=min(mod1_res.fittedvalues)
xmax = max(mod1_res.fittedvalues)
plt.hlines(y=0,xmin=xmin*0.9,xmax=xmax*1.1,color='red',linestyle='--',lw=3)
plt.xlabel("Fitted values",fontsize=15)
plt.ylabel("Residuals",fontsize=15)
plt.title("Fitted vs. residuals plot",fontsize=18)
plt.grid(True)
plt.show()
```



1.19 Standardized regression coefficients

S tandardized regression coefficients provide an easy way to estimate effect size that is independent of units.

Although extracting standardized coefficients is farily easy in R, we have to be a bit more explicit in Python: 1. Transform the Y and each column of the X matrices into standardize values (z-scores) with mean = 0 and standard deviation = 1.0. 2. Run the regression with the standardized inputs. This provides standardized regression coefficients 3. Extract and display the standardized coefficient

1.19.1 Creating standardized input matrices

We use the `zscore()` method from Scipy. The only trick is that `zscore()` returns an array and we prefer to work with Pandas data frames (or series, for single-column data frames). To get around this, we wrap the `zscore()` call inside the `Series()` constructor. We pass the constructor the name of the original Y series to keep everything the same.

[33]:
```
from scipy import stats
Y_norm = pd.Series(stats.zscore(Y), name=Y.name)
Y_norm.head(3)
```

[33]:
```
0    0.233336
1   -0.061669
```

```
2    0.283264
Name: Strength, dtype: float64
```

The X matrix is a bit trickier because the first column (the "const" column we created above) has zero variance—recall that it is just a column of 1's. The definition of z-score is

$$z = \frac{x - \bar{x}}{S}.$$

If there is no variance, the z-score is undefined and everything breaks. To get around this, we do the following: 1. Create a new data frame called "X1_norm by using the Pandas `loc[]` function to select just a subset of columns. In the first line, I select all rows (:) and all columns where the column name is not equal to"const. 2. Apply the `zscore()` method to the entire "X1_norm" data frame. 3. Since we stripped the constant in the first line, add it back by recalling the `add_constant()` method 4. I apply the column names from my original "X1" data frame to the new "X1_norm" data frame 5. Perform a quick check to confirm the values for all explanatory variables are normalized with mean = 0 and (population) standard deviation = 1.

[34]:
```
X1_norm = X1.loc[:, X1.columns != "const"]
X1_norm = pd.DataFrame(stats.zscore(X1_norm))
X1_norm = sm.add_constant(X1_norm)
X1_norm.columns = X1.columns
check = pd.concat([round(X1_norm.mean(axis=0), 5), round(X1_norm.std(axis=0,␣
 ↪ddof=0), 5)], axis=1)
check.columns=["mean", "std dev"]
check
```

[34]:

	mean	std dev
const	1.0	0.0
No	-0.0	1.0
Cement	0.0	1.0
Slag	0.0	1.0
FlyAsh	0.0	1.0
Water	-0.0	1.0
SP	0.0	1.0
CoarseAgg	0.0	1.0
AirEntrain_Yes	0.0	1.0

1.19.2 Running the standardized regression

Once the standardized input matrices are in place, running a standardized regression is no different from running any other regression. The difference is that we know the coefficients are now expressed in terms of the number of standard deviations rather than kilograms, megapascals, and so on.

[35]:
```
modstd = sm.OLS(Y_norm, X1_norm)
modstd_res = modstd.fit()
modstd_res.summary()
```

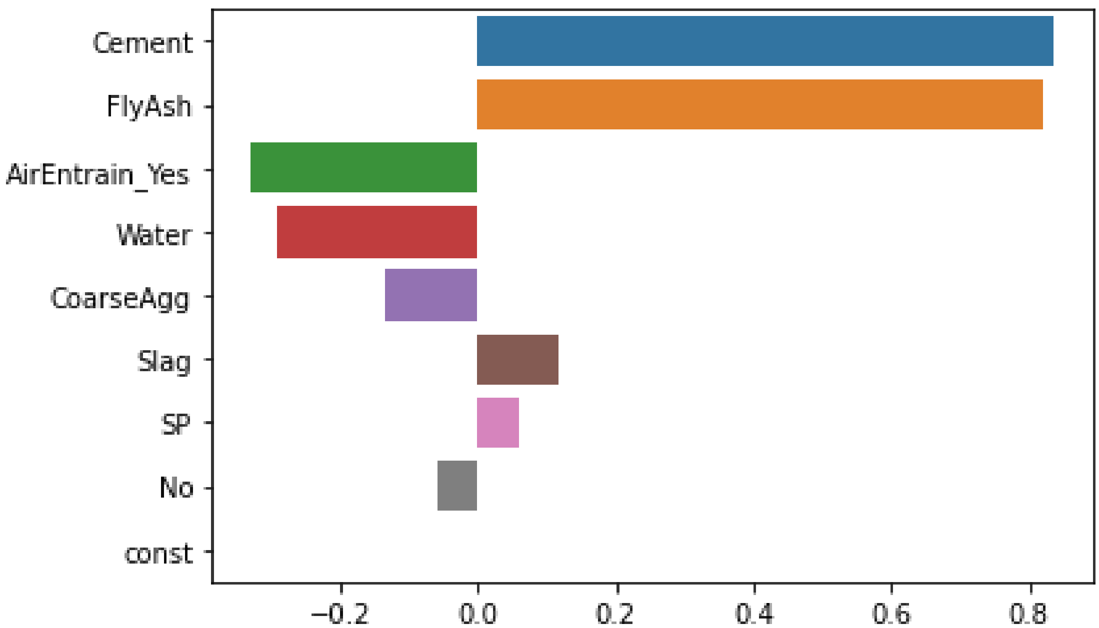
[35]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:               Strength   R-squared:                       0.924
Model:                            OLS   Adj. R-squared:                  0.917
Method:                 Least Squares   F-statistic:                     142.2
Date:                Sun, 10 Sep 2023   Prob (F-statistic):           4.73e-49
Time:                        12:21:53   Log-Likelihood:                -13.650
No. Observations:                 103   AIC:                             45.30
Df Residuals:                      94   BIC:                             69.01
Df Model:                           8
Covariance Type:            nonrobust
================================================================================
==
                     coef    std err          t      P>|t|      [0.025
0.975]
--------------------------------------------------------------------------------
--
const           1.947e-16      0.028   6.83e-15      1.000      -0.057
0.057
No                -0.0575      0.034     -1.674      0.097      -0.126
0.011
Cement             0.8336      0.046     18.070      0.000       0.742
0.925
Slag               0.1175      0.042      2.819      0.006       0.035
0.200
FlyAsh             0.8180      0.047     17.367      0.000       0.724
0.911
Water             -0.2903      0.041     -7.131      0.000      -0.371
-0.209
SP                 0.0591      0.033      1.791      0.077      -0.006
0.125
CoarseAgg         -0.1342      0.045     -2.964      0.004      -0.224
-0.044
AirEntrain_Yes    -0.3282      0.030    -10.946      0.000      -0.388
-0.269
==============================================================================
Omnibus:                        4.255   Durbin-Watson:                   1.637
Prob(Omnibus):                  0.119   Jarque-Bera (JB):                3.680
Skew:                           0.352   Prob(JB):                        0.159
Kurtosis:                       3.601   Cond. No.                         4.11
==============================================================================

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

1.19.3 Tornado diagram

Once we have the regression results, we can extract the coefficients using the `params` property and graph the standardized coefficients. The only trick to getting a tornado diagram is that the coefficients have to be sorted in descending order by the *absolute value* of the coefficient. We resort to a bit of Python trickery to get the items in the desired order. As before, we see that "Cement" and "FlyAsh" are the most important drivers of concrete strength.

[36]:
```
coeff = modstd_res.params
coeff = coeff.iloc[(coeff.abs()*-1.0).argsort()]
sns.barplot(x=coeff.values, y=coeff.index, orient='h');
```



1.19.4 Conclusion

1. As before, we see that "Cement" and "FlyAsh" are the most important drivers of concrete strength.
2. The fitted model can now be used as a **surrogate** for analyzing, evaluating and predicting compressive strength for other compositions of the basic ingredients, as well as those of the process itself.

This is an excellent example of surrogate modeling in a real scientific context.

[ ]:
```
```