COSC757 Assignment 1

Devere Anthony Weaver
Towson University
Towson, USA
dweave8@students.towson.edu

I. Introduction

In a 1998 paper [1], author Yeh uses a dataset to build a model predict the strength of high-performance concrete (HPC). The paper discusses the complexity of modeling the behavior of high-performance concrete (HPC). Evidently, HPC differs from conventional concrete because it includes supplementary cementitious materials like fly ash and blast furnace slag, along with chemical admixtures such as superplasticizers.

While the strength of concrete has historically been associated with the water-to-cement (w/c) ratio, it has become clear that this relationship is not as straightforward, especially for HPC. Various ingredients significantly influence the strength beyond the w/c ratio.

The Abrams' rule, which links the w/c ratio to concrete strength, is revisited. Although it remains relevant, deviations in concrete strength can occur even with identical w/c ratios if other ingredients vary. Therefore, using empirical equations based solely on w/c ratio is insufficient for HPC, which calls for a more sophisticated approach to model its strength.

The author discusses the architecture of an Artificial Neural Network (ANN), which he then aims to use to predict the compressive strength of different mixtures using experimental data.

II. Dataset Description

The "Data Sets" section of the paper describes the experimental data used to verify the reliability of the strength model developed using artificial neural networks (ANN).

The data comes from 17 different sources and includes 1,030 samples of high-performance concrete (HPC) mixes. However, some samples were excluded because they used larger aggregates or had special curing conditions, leaving 727 samples of concrete made with ordinary Portland cement.

Each sample in the dataset consists of eight input variables (cement, fly ash, blast furnace slag, water, superplasticizer, coarse aggregate, fine aggregate, and the age at which strength is tested) and one output variable (compressive strength). In the paper, the dataset was split into four parts (sets A, B, C, and D) to test the model's accuracy. For each test, three sets were used for training the ANN model, and the remaining set was used to test the model's performance.

While the paper states the eight input variables and one target variable, their descriptions are never explicitly explained, which I presume means the user of the dataset must understand the cement mixing process. I do not. Also, while the paper only ended up using 737 samples, I ended up using the entire dataset, all 1,030 samples, as I'm not sure what the criteria for choosing the individual observations was in the original paper.

III. Exploratory Data Analysis

Before any modeling or testing, I first perform some exploratory data analysis. After importing the dataset, I first renamed all the variable names and got an overview of the total number of observations, data types, and any missing value counts.

```
RangeIndex: 1030 entries, 0 to 1029
Data columns (total 9 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Cement                1030 non-null   float64
 1   Blast_Furnance_Slag   1030 non-null   float64
 2   Fly_Ash               1030 non-null   float64
 3   Water                 1030 non-null   float64
 4   Superplasticizer      1030 non-null   float64
 5   Coarse_Aggregate      1030 non-null   float64
 6   Fine_Aggregate        1030 non-null   float64
 7   Age                   1030 non-null   int64
 8   Compressive_Strength  1030 non-null   float64
dtypes: float64(8), int64(1)
```

The above output confirmed there are 1,030 observations with no missing values. Also each of the variables, including the predicted value, are all continuous numerical values.

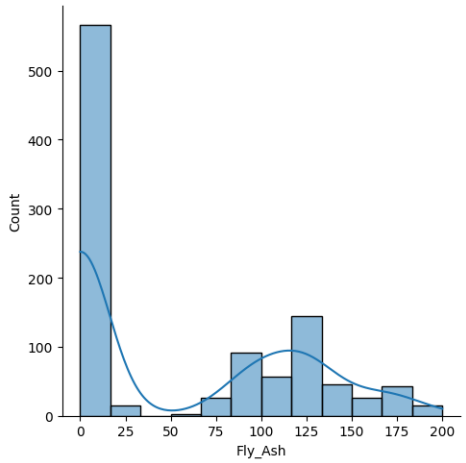
A. Numerical Summary

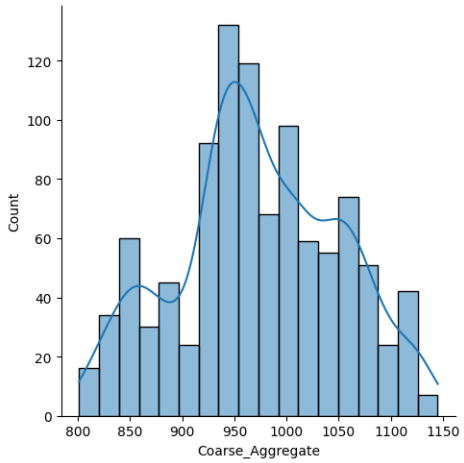
The following graphic shows the numerical five-number summary of each variable.

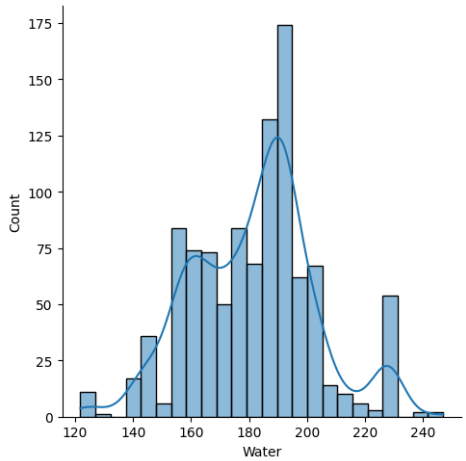
	Cement	Blast_Furnance_Slag	Fly_Ash	Water	Superplasticizer	Coarse_Aggregate	Fine_Aggregate	Age	Compressive_Strength
count	1030.000000	1030.000000	1030.000000	1030.000000	1030.000000	1030.000000	1030.000000	1030.000000	1030.000000
mean	281.167864	73.895825	54.188350	181.567282	6.204660	972.918932	773.580485	45.662136	35.817961
std	104.506364	86.279342	63.997004	21.354219	5.973841	77.753954	80.175980	63.169912	16.705742
min	102.000000	0.000000	0.000000	121.800000	0.000000	801.000000	594.000000	1.000000	2.330000
25%	192.375000	0.000000	0.000000	164.900000	0.000000	932.000000	730.950000	7.000000	23.710000
50%	272.900000	22.000000	0.000000	185.000000	6.400000	968.000000	779.500000	28.000000	34.445000
75%	350.000000	142.950000	118.300000	192.000000	10.200000	1029.400000	824.000000	56.000000	46.135000
max	540.000000	359.400000	200.100000	247.000000	32.200000	1145.000000	992.600000	365.000000	82.600000

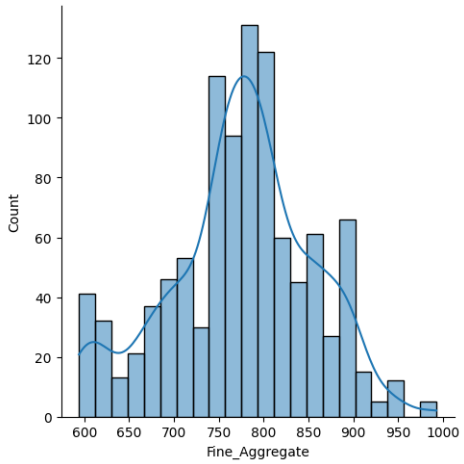
B. Univariate Plots

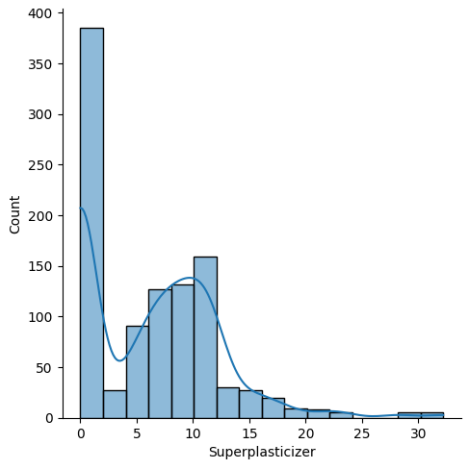
To exam the data further, I generated univariate plots for each of the variables to determine their experimental distribution.

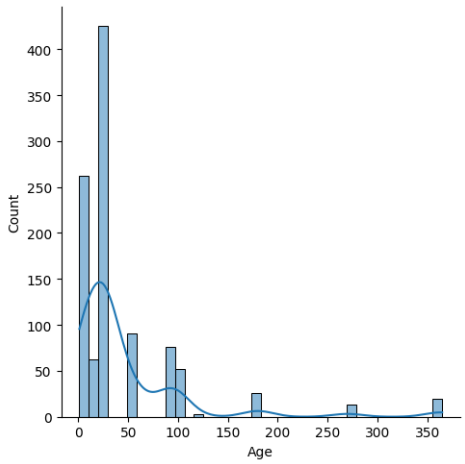

Count
Fly_Ash

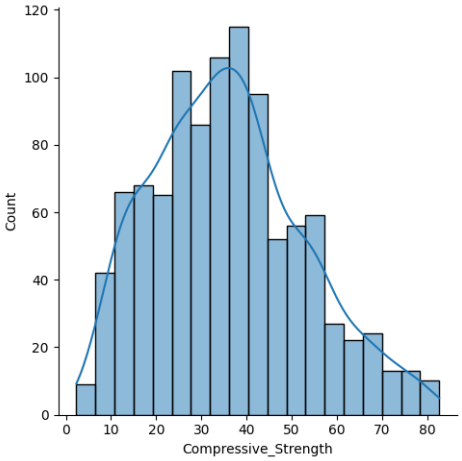

Count
Coarse_Aggregate


Count
Water


Count
Fine_Aggregate


Count
Superplasticizer


Count
Age



Initial inspection of each plot shows that each appears to have a unique distribution. In particular, none appear normal.

C. Bivariate Relationships

To see if there exists any relationship between the any of the variables, I generated a scatter plot and a correlation matrix. (See Appendix for scatter plot and correlation matrix)

IV. Data Preprocessing

To do some preprocessing of the data I performed some normality test, transformations, and tried different binning schemes for one of the continuous variables.

A. Tests of Normality

Graphically, none of the variables in the dataset appear to be normally distributed. However, we shouldn't only rely on graphical tools, we should also use support findings with statistical tests as well. Doing so will improve our analysis.

To help support our visual evidence, we'll also conduct a few statistical tests for normality. These include:

1) Shapiro-Wilk test
2) Kolmogorov-Smirnov test

These two tests are used often in statistical inference to determine whether variables empirically come from a normal distribution.

They both have the following hypotheses:

$$H_0 : (X_1, X_2 \ldots, X_n) \sim Norm(\mu, \sigma)$$

$$H_A : (X_1, X_2 \ldots, X_n) \nsim Norm(\mu, \sigma)$$

where $X_1, X_2, \ldots, X_n$ are samples of a given variable.

Thus, computationally, a "small" p-value means that we'll assume the data is not normally distributed and a "large" p-value means that we'll assume the data is normally distributed. Both are computed differently; however, the results for each test will *generally* agree, although not always.

```
Tests of Normality:
Cement:
 ShapiroResult(statistic=0.9589606405907329, pvalue=2.0816997308487217e-16)
 KstestResult(statistic=1.0, pvalue=0.0, statistic_location=102.0, statistic_sign=-1)

Blast_Furnance_Slag:
 ShapiroResult(statistic=0.8124091681661155, pvalue=5.795509854833019e-33)
 KstestResult(statistic=0.5427184466019417, pvalue=9.766338588352471e-285, statistic_location=11.0, statistic_sign=-1)

Fly_Ash:
 ShapiroResult(statistic=0.7619989436807126, pvalue=4.1346984419547427e-36)
 KstestResult(statistic=0.5, pvalue=1.2357510751860894e-238, statistic_location=0.0, statistic_sign=-1)

Water:
 ShapiroResult(statistic=0.9803903374425459, pvalue=1.4629769256128603e-10)
 KstestResult(statistic=1.0, pvalue=0.0, statistic_location=121.8, statistic_sign=-1)

Superplasticizer:
 ShapiroResult(statistic=0.8660308133288318, pvalue=9.064747206687283e-29)
 KstestResult(statistic=0.6219510728421562, pvalue=0.0, statistic_location=3.0, statistic_sign=-1)

Coarse_Aggregate:
 ShapiroResult(statistic=0.9824530513495914, pvalue=8.347134903632559e-10)
 KstestResult(statistic=1.0, pvalue=0.0, statistic_location=801.0, statistic_sign=-1)

Fine_Aggregate:
 ShapiroResult(statistic=0.9806713540354409, pvalue=1.8419600200281133e-10)
 KstestResult(statistic=1.0, pvalue=0.0, statistic_location=594.0, statistic_sign=-1)

Age:
 ShapiroResult(statistic=0.5907055333539084, pvalue=7.718418581998338e-44)
 KstestResult(statistic=0.9967083543955544, pvalue=0.0, statistic_location=3, statistic_sign=-1)

Compressive_Strength:
 ShapiroResult(statistic=0.9797907202264295, pvalue=9.010082871702769e-11)
 KstestResult(statistic=0.998579038973, pvalue=0.0, statistic_location=3.32, statistic_sign=-1)
```

Observe the test results for each variable. The small p-values indicate we should assume the variables aren't normally distributed, which supports our graphical evidence.

B. Normalization

There exists methods to normalize our data. In this section, I'll describe the three methods I used for each continuous variable:

1) min-max normalization
2) z-score normalization
3) decimal scaling normalization

For the following subsections, let X refer to the original value and X^* refer to the normalized field value.

1) Min-max normalization: Min-max normalization works by seeing how much greater the field value is than the minimum value $min(X)$, and scaling this difference by the range.

$$X^* = \frac{X - min(X)}{range(X)} = \frac{X - min(X)}{max(X) - min(X)} \tag{1}$$

The data values that represent the minimum for the variable will have a min-max normalization value of zero. The data values that represent the midrange data value has a min-max normalization of 0.5. The data values representing the field maximum will have a min-max normalization of 1. Thus, the min-max normalization will range from 0 to 1.

To do perform this normalization on the entire dataset, I implemented a function using the following definition:

```
[ ]  # function that will min-max normalize the input data
     def min_max(x):
       range = max(x) - min(x)
       return (x - min(x))/(range)
```

After vectorizing the function, I created a new dataframe with the min-max version of the variables.

	Cement	Blast_Furnance_Slag	Fly_Ash	Water	Superplasticizer	Coarse_Aggregate	Fine_Aggregate	Age	Compressive_Strength
0	1.000000	0.000000	0.0	0.321086	0.07764	0.694767	0.205720	0.074176	0.967485
1	1.000000	0.000000	0.0	0.321086	0.07764	0.738372	0.205720	0.074176	0.741996
2	0.526256	0.396494	0.0	0.848243	0.00000	0.380814	0.000000	0.739011	0.472655
3	0.526256	0.396494	0.0	0.848243	0.00000	0.380814	0.000000	1.000000	0.482372
4	0.220548	0.368392	0.0	0.560703	0.00000	0.515698	0.580783	0.986264	0.522860

2) Z-score standardization: Z-score normalization works by taking the difference between the field value and the field mean value, and scaling this difference by the standard deviation of the field values.

$$Z - score = \frac{X - \mu_X}{\sigma_X} \tag{2}$$

The data values that lie below the mean will have a negative Z-score standardization. The values falling exactly on the mean will have a Z-score standardization of zero. Data values that lie above the mean will have a positive Z-score standardization.

To perform this normalization, I implemented a function using the following definition:

```
[ ]  # z-score function
     def z_score(x):
       mean = np.mean(x)
       sd = np.std(x)
       return (x - mean)/sd
```

The result is a new dataframe with the normalized variables after vectorizing the function. The following is a sample from this new table.

	Cement	Blast_Furnance_Slag	Fly_Ash	Water
0	2.477915	-0.856888	-0.847144	-0.916764
1	2.477915	-0.856888	-0.847144	-0.916764
2	0.491425	0.795526	-0.847144	2.175461
3	0.491425	0.795526	-0.847144	2.175461
4	-0.790459	0.678408	-0.847144	0.488793

3) Decimal Scaling: Decimal scaling ensures that every normalized values lies between -1 and 1.

$$X^*_{decimal} = \frac{X}{10^d} \tag{3}$$

where d represents the number of digits in the data value with the largest absolute value. I implemented this method using the following function.

```
[ ]  def decimal_scale(x):
       a = np.abs(x)
       a = np.max(x)
       a_str = str(a)
       d = len(a_str) - 1
       return (x)/(10**d)
```

The following is a snippet from this new dataframe.

	Cement	Blast_Furnance_Slag	Fly_Ash	Water
0	0.05400	0.00000	0.0	0.0162
1	0.05400	0.00000	0.0	0.0162
2	0.03325	0.01425	0.0	0.0228
3	0.03325	0.01425	0.0	0.0228
4	0.01986	0.01324	0.0	0.0192

C. Binning a Continuous Variable

We'll try different methods of binning one of our continuous variables.
For our choice, we'll use

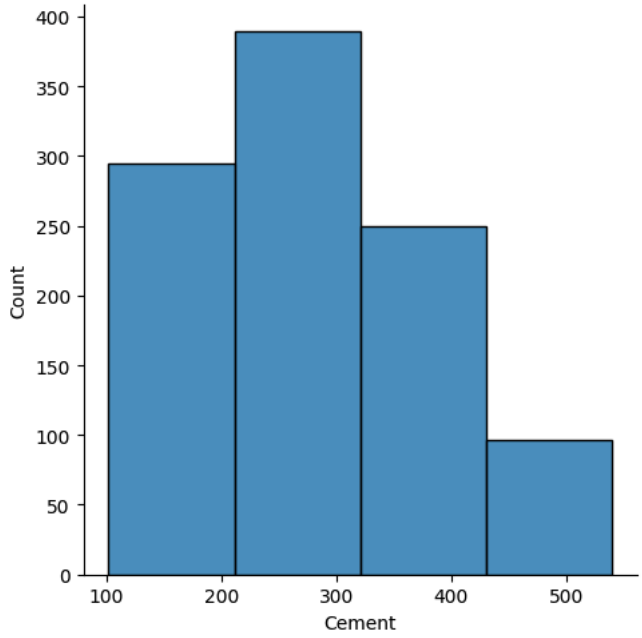
1) Equal width binning
2) Equal frequency binning

The continuous variable of choice is will be 'Concrete'.

To do equal width binning, dividing the numerical predictor into k categories of equal width, where k is chosen by the analyst. I tried it with k=100 and k=50 to see what this looks like.

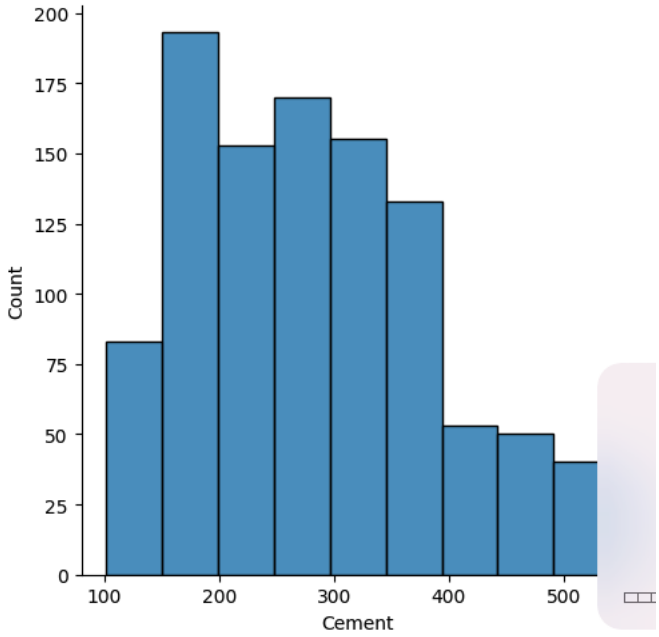
```
# k = 100
sns.displot(data['Cement'], binwidth = 100)
```

```
<seaborn.axisgrid.FacetGrid at 0x7c74b6996260>
```



```
# k = 50
sns.displot(data['Cement'], binwidth = 50)
--VISUAL--
```

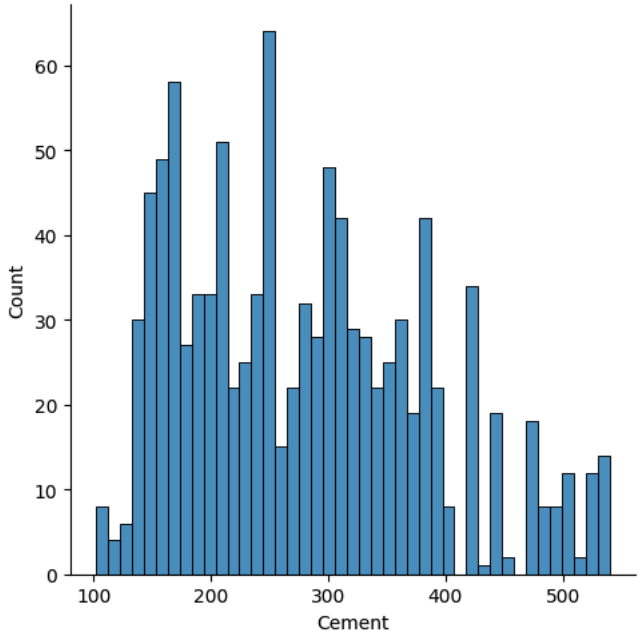
```
<seaborn.axisgrid.FacetGrid at 0x7c74b6997820>
```



Equal frequency binning - dividing the numerical predictor into k categories each having n/k records, where n is the total number of records. Again, we'll try it with k=100 and k=50 to observe the difference in effect. n will be the number of total records and in this case it's 1030.

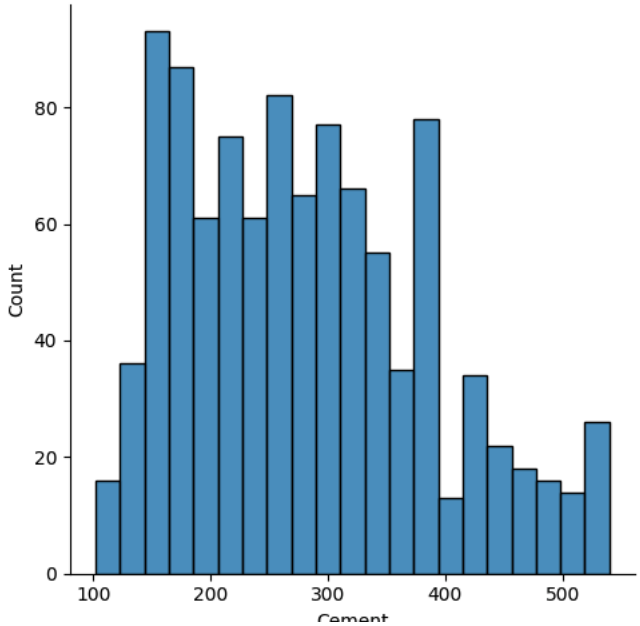
```
# k = 100
bins = 1030/100
sns.displot(data['Cement'], binwidth = bins)
```

```
<seaborn.axisgrid.FacetGrid at 0x7c74ba271c90>
```




```
# k = 50
bins = 1030/50
sns.displot(data['Cement'], binwidth = bins)
```

```
<seaborn.axisgrid.FacetGrid at 0x7c74b3e82800>
```



D. Transformation

To demonstrate how to use transformations on continuous variables, I used the following transformations on the 'Cement' variable.

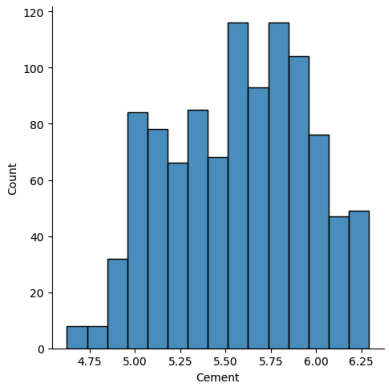
1) Natural log
2) Square root
3) Inverse square root

1) Natural Log:

```
# natural-log transformation
natural_log_tx = np.log(data['Cement'])

# check for normality graphically and numerically
sns.displot(natural_log_tx)
print(f"Cement: \n {shapiro(natural_log_tx)} \n {kstest(natural_log_tx, 'norm')} \n")
```

```
Cement:
 ShapiroResult(statistic=0.9783841893340678, pvalue=2.9916946895132117e-11)
 KstestResult(statistic=0.9999981267623093, pvalue=0.0, statistic_location=4.62497281
```

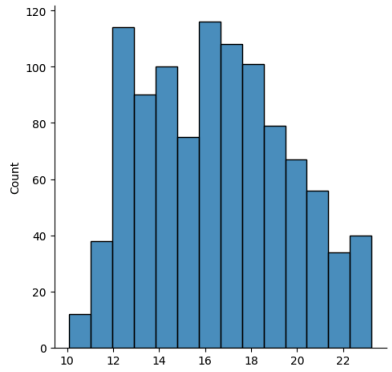


2) Square Root:

```
# sqrt transformation
sqrt_tx = np.sqrt(data['Cement'])

# check for normality graphically and numerically
sns.displot(sqrt_tx)
print(f"Cement: \n {shapiro(sqrt_tx)} \n {kstest(sqrt_tx, 'norm')} \n")
--INSERT--
```

```
Cement:
 ShapiroResult(statistic=0.9765290852721249, pvalue=7.479452946366543e-12)
 KstestResult(statistic=1.0, pvalue=0.0, statistic_location=10.099504938362077,
```

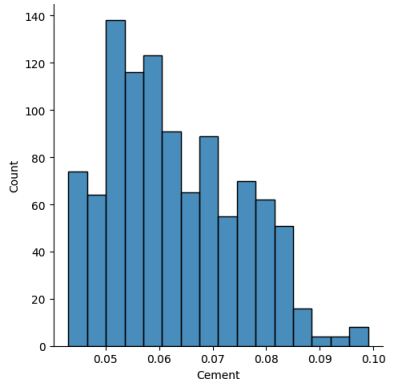


3) Inverse Square Root:

```
# invesrse sqrt transformation
inverse_sqrt_tx = data['Cement']**(-1/2)

# check for normality graphically and numerically
sns.displot(inverse_sqrt_tx)
print(f"Cement: \n {shapiro(inverse_sqrt_tx)} \n {kstest(inverse_sqrt_tx, 'norm')} \n")
--NORMAL--
```

```
Cement:
 ShapiroResult(statistic=0.9639501172848446, pvalue=2.8074053663722047e-15)
 KstestResult(statistic=0.5171624450977758, pvalue=1.522780875850195e-256, statistic_loc
```



V. Regression Analysis

In the original paper, the author used an ANN for regression. The results of the paper showed that this regression model had increased performance over more traditional statistical models. However, since I'm not familiar with building neural networks, I just used a trivial multiple regression and compared it with the results of the paper. Ultimately, the linear model I built had the same metrics as the comparison model in the paper.

To help determine which variables to use in the model I reviewed the EDA phase and determined the results weren't particularly helpful in determining the "best" variables for the regression model. Thus, I decided to use forward step-wise regression to determine which variables to use.

I used some libraries that implement this algorithm to determine which variables built the best model.

```
import pandas as pd
import numpy as np
from sklearn import linear_model
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from mlxtend.feature_selection import SequentialFeatureSelector

# select features
features = data.iloc[:, :-1]
features

# select target
target = data.iloc[:, -1]
target

# stepwise regresssion
sfs = SequentialFeatureSelector(linear_model.LinearRegression(),
                                k_features=8,
                                forward = True,
                                scoring = "neg_mean_squared_error",
                                cv = None)

# fit select features
selected_features = sfs.fit(features, target)

# check which features were used
selected_features.k_feature_names_
--INSERT--
```

```
('Cement',
 'Blast_Furnance_Slag',
 'Fly_Ash',
 'Water',
 'Superplasticizer',
 'Coarse_Aggregate',
 'Fine_Aggregate',
 'Age')
```

The algorithm determined that all input variables should be used for this model. This confirms my results I received by building the model and adding variables manually.

To build the model and test the performance, again, I split the dataset into a testing and training set and built a new dataset with the predicted values along with the actual values.

```
# do the test/train split thing
x_train, x_test, y_train, y_test = train_test_split(
    data.iloc[:, :-1],
    target,
    test_size = .2,
)

# fit model to data using selected features
lr = linear_model.LinearRegression()
lr.fit(x_train, y_train)

# check out model
pred = lr.predict(x_test)

# create data frame to check results
results = pd.DataFrame()
results['predicted'] = pred
results['actual'] = y_test.reset_index(drop = True)
results
```

--INSERT--

	predicted	actual
0	28.439268	31.35
1	47.526967	54.28
2	56.774359	77.30
3	30.948934	14.54
4	58.000955	41.15
...	...	...
201	51.920399	69.84
202	40.817275	33.76
203	24.217973	24.99
204	23.888102	26.85
205	13.388881	9.31

206 rows × 2 columns

After building this new dataframe, I computed a few different metrics that are commonly used when judging model performance.

```
[ ] # check out some metrics, start with R^2
    from sklearn.metrics import r2_score
    score = r2_score(results["actual"], results["predicted"])
    print(f"R^2 score: {score}")
```

```
R^2 score: 0.6807846463995304
```

```
[ ] # MAE
    from sklearn.metrics import mean_absolute_error
    score = mean_absolute_error(results["actual"], results["predicted"])
    print(f"MAE score: {score}")
```

```
MAE score: 7.890393921177814
```

```
[ ] # RMSE
    from sklearn.metrics import mean_squared_error
    score = mean_squared_error(results["actual"], results["predicted"])
    print(f"MSE score: {score}")
```

```
MSE score: 93.40627956087397
```

Overall, these metrics indicate the model is fairly poor for prediction purposed. It's not one that I would personally consider worth using, and this is even before attempting to look at any model diagnostics.

VI. Conclusion

Overall, this lab was incredibly helpful getting some hands-on practice with the EDA process and data transformations. The model is very poor; however, this is most likely due to the complexity of the relationships, if any, amongst the variables. Hence, why the author of the original paper argued in favor of using neural networks in an attempt to capture this complexity.

References

[1] I.-C. Yeh, "Modeling of strength of high-performance concrete using artificial neural networks," Cement and Concrete Research, vol. 28, no. 12, pp. 1797–1808, Dec. 1998, doi: 10.1016/S0008-8846(98)00165-3.

VII. Appendix

Fig. 1. Scatter Matrix

	Cement	Blast_Furnance_Slag	Fly_Ash	Water	Superplasticizer	Coarse_Aggregate	Fine_Aggregate	Age	Compressive_Strength
Cement	1.000000	-0.275216	-0.397467	-0.081587	0.092386	-0.109349	-0.222718	0.081946	0.497832
Blast_Furnance_Slag	-0.275216	1.000000	-0.323580	0.107252	0.043270	-0.283999	-0.281603	-0.044246	0.134829
Fly_Ash	-0.397467	-0.323580	1.000000	-0.256984	0.377503	-0.009961	0.079108	-0.154371	-0.105755
Water	-0.081587	0.107252	-0.256984	1.000000	-0.657533	-0.182294	-0.450661	0.277618	-0.289633
Superplasticizer	0.092386	0.043270	0.377503	-0.657533	1.000000	-0.265999	0.222691	-0.192700	0.366079
Coarse_Aggregate	-0.109349	-0.283999	-0.009961	-0.182294	-0.265999	1.000000	-0.178481	-0.003016	-0.164935
Fine_Aggregate	-0.222718	-0.281603	0.079108	-0.450661	0.222691	-0.178481	1.000000	-0.156095	-0.167241
Age	0.081946	-0.044246	-0.154371	0.277618	-0.192700	-0.003016	-0.156095	1.000000	0.328873
Compressive_Strength	0.497832	0.134829	-0.105755	-0.289633	0.366079	-0.164935	-0.167241	0.328873	1.000000

Fig. 2. Correlation Matrix