Data Pre-Processing

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

Preliminaries

Packages

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

```
    Analysis on 2021-06-25 at 01:11
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

Read

```
#d = pd.read_excel('data/employee.xlsx')
d = pd.read_excel('http://lessRstats.com/data/employee.xlsx')
```

The data values in the Name column are not values per se to analyze, but instead serve as row identifiers, ID's. As such, replace the default integer row labels with the values of the column Name. Do so with the `set_index()` function.

```
d = d.set_index('Name')
d.head()
```

	Years	Gender	Dept	Salary	JobSat	Plan	Pre	Post
Name								
Ritchie, Darnell	7.0	M	ADMN	53788.26	med	1	82	92
Wu, James	NaN	M	SALE	94494.58	low	1	62	74
Hoang, Binh	15.0	M	SALE	111074.86	low	3	96	97
Jones, Alissa	5.0	F	NaN	53772.58	NaN	1	65	62
Downs, Deborah	7.0	F	FINC	57139.90	high	2	90	86

Create Dummy Variables

designateMachine learning procedures cannot directly process categorical variables with non-numeric values. For example, consider a data set with two values of *Gender*, coded as M and F. The variable *Gender* with this non-numerical coding cannot be entered into a machine learning analysis, which requires numerical variables only.

Categorical variables with non-numerical values, however, can be converted to numerical representations. Many such conversions are possible. Here we consider the most widely used conversion.

> *Dummy Variable*: A numerically encoded variable for each level of a categorical variable, with a value of 1 if the level is present and 0 if not.

The Gender variable becomes two dummy variables, *Gender_F* and *Gender_M*. For example, if a person's Gender is listed as F, then *Gender_F* is 0 and *Gender_M* is 1.

Pandas provides the function `get_dummies()` to covert a categorical variable to a corresponding set of dummy values, one for each category. The parameter `columns` designates the variables to be converted.

One adjustment is needed. If you know the value of *Gender_F* for an individual is 1, then you also know that *Gender_M* is 0. So the value of either one of two dummy variables implies the value of the other. To avoid redundancy, in general, for k levels of the categorical variables, the number of dummy variables retained in the analysis is $k-1$. For two levels of *Gender*, arbitrarily retain $2-1=1$ of the dummy variables in the analysis.

With `get_dummies()`, drop the first dummy variable with the `drop_first` parameter set to `True`. Alphabetically, F comes before M, so in the following analysis, *Gender_F* is dropped. The original *Gender* variable is replaced with *Gender_M*.

For *JobSat* with three levels -- High, Low, and Med -- create three dummy variables, each corresponding to the one of the three values. *JobSat* is then replaced by two dummy variables for the Low and Med values. For example, if you know the values of *JobSat_low* and *JobSat_med* are both 0, then you know that the value of *JobSat_High* is 1. Knowing two values implies the third, so retain only two dummy variables for *JobSat* in the analysis.

```
d = pd.get_dummies(d, columns=["Gender", "JobSat"], drop_first=True)
d.head()
```

	Years	Dept	Salary	Plan	Pre	Post	Gender_M	JobSat_low	JobSat_med
Name									
Ritchie, Darnell	7.0	ADMN	53788.26	1	82	92	1	0	1
Wu, James	NaN	SALE	94494.58	1	62	74	1	1	0
Hoang, Binh	15.0	SALE	111074.86	3	96	97	1	1	0
Jones, Alissa	5.0	NaN	53772.58	1	65	62	0	0	0
Downs, Deborah	7.0	FINC	57139.90	2	90	86	0	0	0

Usually which particular dummy variable is dropped for each categorical variable is irrelevant. If, however, it is desired to drop a dummy variable other than the first, then run `get_dummies()` without the `drop_first` parameter and manually drop the specified dummy variable from the data frame.

Missing Data

Assess Amount of Missing Data

Machine learning functions generally do *not* work in the presence of missing data. Before machine learning analysis, examine the data for missing data and adjust accordingly, either delete the row or column or impute the value.

A missing data value is indicated by the notation `NaN`, an abbreviation for Not a Number. Sometimes functions or discussion of missing data refer to missing data as `na`, which means Not Available.

Here James Wu has a missing value for the number of years he worked at the company. Data values for James Wu occupy the second row of data, identified by row index 1. The row definition of 1:2 (confusingly) also refers to the second row. However, specifying the row as a range results in the output's more visually appealing horizontal placement.

```
d.iloc[1:2, 0:5]
```

	Years	Dept	Salary	Plan	Pre
Name					
Wu, James	NaN	SALE	94494.58	1	62

The `isna()` function indicates if a data value is missing. Follow with the `sum()` function to sum the number of missing values for a variable, here all variables in the *d* data frame because no specific variable is specified. Follow with a second `sum()` function to sum the sums, that is, the total number of missing values in the entire data frame.

```
print (d.isna().sum())
print ('\nTotal Missing:', d.isna().sum().sum())
```

```
    Years         1
    Dept          1
    Salary        0
    Plan          0
    Pre           0
    Post          0
    Gender_M      0
    JobSat_low    0
    JobSat_med    0
    dtype: int64

    Total Missing: 2
```

As a programming note, without using the `print()` function, the last row of code in a Jupyter cell that specifies output generates the default output. If there is more than a single line of code that generates output, or if customization of the output is desired, such as adding a descriptive label, then invoke `print()`, as in this example.

Show Rows with Missing Data

The code for viewing all rows of missing data begins with the `isna()` function, which returns `True` if a data value is missing. The `any()` function evaluates the data frame column-by-column and then returns `True` if there are any `True` values in the corresponding row. Putting the expression within `d[ ]` selects only the rows with `True`, that is, with missing according to `isna()`.

```
 d[d.isna().any(axis='columns')]
```

	Years	Dept	Salary	Plan	Pre	Post	Gender_M	JobSat_low	JobSat_med
Name									
Wu, James	NaN	SALE	94494.58	1	62	74	1	1	0
Jones, Alissa	5.0	NaN	53772.58	1	65	62	0	0	0

Delete rows with Missing Data

The simplest method to address missing data deletes a row if it contains any missing data, what is called *case deletion*, or *list-wise deletion*. The `dropna()` function deletes rows with missing data from *d*. It is also possible to apply the function to columns with parameter `axis`, which indicates if the analysis applies to rows or columns. Often in Python coding people use 0 instead of the more descriptive 'rows'.

The process in this example removes the three rows with missing data, from 37 rows to 34 rows.

```
d.shape
```

```
    (37, 9)
```

```
d = d.dropna()
d.shape
```

```
    (35, 9)
```

The problem with dropping rows that contain missing data is that for some data sets much or most of the data can be deleted. Appropriate if many data values in an entire row are missing, but perhaps not if just one missing data value across data for many variables. Or, sometimes a single variable may contain many missing values, so better to delete the bad variable than delete so may corresponding rows of data (cases).

To illustrate, re-display the variable names and the first five rows of data. In the original data frame, James Wu is missing Years worked for the company, and Alissa Jones is missing the Dept worked in as well and the Job Satisfaction rating. Both rows of data are now deleted from the revised data frame.

```
d.head()
```

	Years	Dept	Salary	Plan	Pre	Post	Gender_M	JobSat_low	JobSat_med
Name									
Ritchie, Darnell	7.0	ADMN	53788.26	1	82	92	1	0	1
Hoang, Binh	15.0	SALE	111074.86	3	96	97	1	1	0
Downs, Deborah	7.0	FINC	57139.90	2	90	86	0	0	0
Afshari, Anbar	6.0	ADMN	69441.93	2	100	100	0	0	0
Knox, Michael	18.0	MKTG	99062.66	3	81	84	1	0	1

Impute Missing Data

A data frame can contain many variables, including variables not relevant to a particular analysis. Later we show that when doing machine learning, we isolate a relevant set of variables for a given model in their own data structure. In machine learning, by tradition name this data structure X, the uppercase representation to indicate more than a single variable in general.

To make this code more applicable to subsequent machine learning analyses, we subset the variables from the original data frame into a data frame named X, the data that contains just the features (predictor variables) for the machine learning analysis.

```
X = d.loc[:, ['Years', 'Salary', 'Pre', 'Post']]
X.head()
```

	Years	Salary	Pre	Post
Name				
Ritchie, Darnell	7.0	53788.26	82	92
Hoang, Binh	15.0	111074.86	96	97
Downs, Deborah	7.0	57139.90	90	86
Afshari, Anbar	6.0	69441.93	100	100
Knox, Michael	18.0	99062.66	81	84

```
type(X)
```

```
    pandas.core.frame.DataFrame
```

Instead of deleting rows or columns with missing data, an alternative approach *imputes* missing data values. Provide some reasonable guess regarding a missing data value and then set this value in place of the missing data code. Impute with the `SimpleImputer()` function. The mean of each variable is the default value to replace missing data for a variable. A typical better choice replaces the mean with the median to avoid the impact of potential outliers.

Specify the median with the `strategy` parameter. The `missing_values` parameter indicates the value defined as a missing value, here `NaN` as indicated by the numpy array value of `nan`. To impute the median, only select variables with numerical values.

Remember, first we have the Python language, then we have the `numpy` package built on top of base Python, then we have the `pandas` package built on top of `numpy`. So when doing machine learning, we encounter functions from the original Python plus from both `numpy` and `pandas`.

The `fit()` function computes the values to be used to fill in missing values. The `transform()` does the imputation.

```
from sklearn.impute import SimpleImputer
imp_med = SimpleImputer(missing_values=np.nan, strategy='median')
imp_med = imp_med.fit(X)
X = imp_med.transform(X)
```

The `transform()` function outputs an array from the original input data frame.

```
type(X)
```

```
    numpy.ndarray
```

The missing data for James Wu for Years now has a value of 9.0, that is, the value in the second row and first column of *d*.

```
X[1,0]
```

```
    15.0
```

The best way to impute missing data involves predicting each missing data value from the remaining non-missing data values. This is more complex and requires much more computer time, so maybe not practical for very large data sets.

Search for Outliers

Sometimes some data values do not appear to part of the same distribution as the other data values.

> *Outlier*: Value considerably different from most remaining values of the distribution.

Note that this definition of an outlier applies to the analysis of a single variable and so identifies what is called a univariate outlier. However, the concept can also be applied to considering several variables simultaneously.

One motivation for outlier detection is that an outlier could represent a simple data collection or transcription error. Alternatively, an outlier could represent a data value sampled from a population distinct from the population that generated the remaining data values and therefore would bias the analysis regarding generalizations to what was intended to be the population of interest. So, an essential aspect of the data analytic process first identifies and then explains the process that generated the anomalous values.

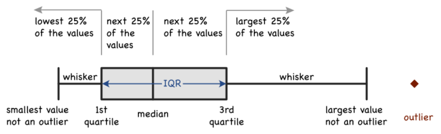
One approach to identify outliers for a single variable follows from the interquartile range (IQR). To compute, sort the values of the distribution from smallest to largest.

> *Quartiles*: Three values that divide the entire sorted distribution of values into quarters, four-equal size groups.

The first one-quarter of the values lie between the smallest value and the first quartile. The second quartile is the median, which occupies the middle spot between the smallest and largest values in the sorted distribution. The third quartile separates the largest 25% of the values from the smaller values.

> *Interquartile Range* (IQR): Range of values that contains the middle 50% of the values in magnitude, the positive difference between the 3rd and 1st quartiles.

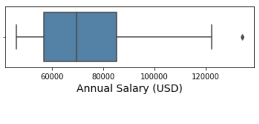
> *Boxplot*: The body of the box extends from approximately the 1st to the 3rd quartiles, with a line through the median and perpendicular lines called whiskers extending out from the edges, with outliers plotted beyond the whiskers.



Values far from the edges of the box are labeled as outliers. The classic definition is that data values beyond 1.5 IQR's and 3 IQR's beyond the 1st and 3rd quartiles are labeled as potential outliers and outliers.

To plot a boxplot with `seaborn`, use the function `boxplot()`. By default, the boxplot is quite high. Can use the `figure` function with the `figsize` parameter to specified a more narrow height.

```
plt.figure(figsize=(6,1.5))
sns.boxplot(x=d['Salary'], color='steelblue')
plt.xlabel('Annual Salary (USD)', fontsize=14)
plt.show()
```



Identify the row of data that has the maximum value for Salary with the `idmax()` function. The `index` is the pandas name for the row names, which in the *d* data frame are the actual employee names, not integers. and display just the value of Salary instead of all the data values for that row (case). (Another approach would be to use filter rows expression from the data wrangingly notebook and list all records larger than about 125,000 or something.)

```
d['Salary'].idxmax()
```

```
    'Correll, Trevon'
```

```
round(d['Salary'].loc[d['Salary'].idxmax()], 2)
```

```
    134419.23
```

The implication for data analysis is to examine the process that generated this anomalous data value. Is the person making that large salary a high-level manager with the other salaries are for hourly workers? If different processes generate the salaries, then the outlier should be deleted from further analysis, and the results of further analyses generalized to the appropriate population, that of hourly workers, in this example.

Transform Variables to Similar Scale

Machine learning algorithms tend to generate more useful results when the variables on which they operate are all on about the same scale. That means the data values for each of the variables have about the same range at least, and maybe even about the same mean and standard deviation. Variables with the values as they originally exist do not generally arrive for analysis sharing the same scale. Weight in pounds, for example, is scaled in entirely different units than Height in inches, or Annual_Income in USD.

Transform the data values of variables with different scaling to obtain similar scales. The re-scalings discussed here are all *linear*.

> *Linear transformation*: Does not change the shape of the underlying distribution, nor its relations with other variables, just its scale.

After the re-scaling, an approximately normal distribution remains approximately normal, and a skewed distribution remains skewed.

For a linear transformation, multiply each data value by a constant and add another constant. That is, transform variable x to y with $y = a + bx$. An example is converting measurement of length in feet to inches in which the data values are divided by 12, that is, $b = 1/12$ and $a = 0$. That is, to convert feet to inches, divide each data value expressed in feet by 12.

The Python package for machine learning, `sklearn`, module `preprocessing`, provides several rescaling possibilities.

```
from sklearn import preprocessing
```

Specify the variables to transform in their own data frame.

As a programming note, subsets (slices) of data frames by default do not create clean copies. Changes to the values of X can lead back to changes in the original data frame from which X was derived. To make X completely independent of *d*, invoke the subset slice with the `copy()` function.

```
X = d[['Years', 'Salary', 'Pre']].copy()
X.loc[:,'Pre'] = X.loc[:,'Pre'].astype('float64')
X.head()
```

	Years	Salary	Pre
Name			
Ritchie, Darnell	7.0	53788.26	82.0
Hoang, Binh	15.0	111074.86	96.0
Downs, Deborah	7.0	57139.90	90.0
Afshari, Anbar	6.0	69441.93	100.0
Knox, Michael	18.0	99062.66	81.0

How do know what (if any) preprocessing methods to choose? There are some guidelines, but the most general answer follows the typical machine learning approach: Try the various possibilities and choose the algorithm that provides the most accurate forecasting.

Suppose we wish to explain a score on a post test from years worked at the company, annual salary, and scores on a corresponding pre-test. In the following examples, isolate those three variables into a data structure called X.

Min-Max Scaling

A common rescaling used in machine learning transforms the variables to have the same minimum and maximum values.

> *Min-Max Scaling*: Convert all data values for a variable so that the minimum value is 0 and the maximum value is 1.

Write this transformation for the value of x in the i^{th} row of data values of variable x as:

$$y_i = \frac{x_i - min(x)}{max(x) - min(x)}$$

Express as a linear function with $a = -(min(x)/range(x))$ and $b = 1/range(x)$.

The `MinMaxScaler` provides the needed transformation from the minimum and maximum values of each variable to which the transformation is applied.

```
from sklearn.preprocessing import MinMaxScaler
mm_scaler = preprocessing.MinMaxScaler()
```

Apply to Original Data

At this point, X is a (pandas) data frame.

```
type(X)
```

```
pandas.core.frame.DataFrame
```

The `fit()` function calculates the needed information to provide the rescaling: the minimum and maximum needed for each variable. The `transform()` function performs the rescaling using this information. Combine both methods with a single invocation of `fit_transform()`.

The `MinMaxScaler` works with and retains missing data. Any entered non-numeric variables, however, cannot be processed and cause the routine to terminate.

```
Xmm = mm_scaler.fit_transform(X)
```

After processing by the `MinMaxScaler` and transformed, X is now a (numpy) array. This is one of the complications of Python applied to data analysis, the need for external packages, here both `numpy` and `pandas`. Unfortunately, sometimes, as in this example, data structures are created that conform to those of another package than what was input. A `numpy` array is the equivalent data structure to a `pandas` data frame.

```
type(Xmm)
```

```
numpy.ndarray
```

Some statistical computations use `panda` data frames instead of `numpy` arrays. The default display of the data frame also is more aesthetic than the array. Convert the transformed X back to a data frame. The column names also have to be manually added as the `numpy` array deletes the original names.

Examine the first five rows of the data frame. The values of each of the variables *Years*, *Salary*, and *Pre* were initially on discordant scales, but now have values that range from 0 to 1.

```
Xmm = pd.DataFrame(Xmm, columns=['Years', 'Salary', 'Pre'])
Xmm.head()
```

	Years	Salary	Pre
0	0.260870	0.086793	0.560976
1	0.608696	0.735607	0.902439
2	0.260870	0.124753	0.756098
3	0.217391	0.264082	1.000000
4	0.739130	0.599560	0.536585

Confirm the success of the transformations of the three variables by using the pandas data frame methods `min` and `max`.

```
Xmm.min()
```

```
Years     0.0
Salary    0.0
Pre       0.0
dtype: float64
```

```
Xmm.max()
```

```
Years     1.0
Salary    1.0
Pre       1.0
dtype: float64
```

Can transform any data with the same values from the previous application of `fit()`. Here manually transform the second row of data in X.

```
mm_scaler.transform([[15.0, 111074.86, 96.0]])
```

```
array([[0.60869565, 0.73560716, 0.90243902]])
```

You can also transform manually. The linear equation by which each data value is transformed is available from the computed `scale_` and `min_` data structures. Each computed data structure contains one value for each variable in the data frame that is transformed.

```
print('Multiplier:', mm_scaler.scale_.round(3))
print('Additive:', mm_scaler.min_.round(3))
```

```
Multiplier: [0.043 0.    0.024]
Additive: [-0.043 -0.522 -1.439]
```

To illustrate, manually compute the transformed value of Salary for the second row of data in X, which equals the corresponding value in the above display portion of the X data frame.

```
mm_scaler.scale_[1]*111074.86 + mm_scaler.min_[1]
```

```
0.7356071617792598
```

Apply to New Data

The purpose of supervised learning is to develop a model for forecasting from the values of the feature variables. If the data have undergone a scaling transformation such as Min-Max, then any new data must undergo that same transformation before generating a forecast.

The new data can be transformed manually, as in the immediately previous example. Or, apply the information already obtained from the `fit()` function encoded in `mm_scaler` to the transformation with the `transform()` function.

In this example, suppose an employee has worked at the company for 8 years, has a salary of 76,492 USD, and scored an 89 on the pre-test. Next, compute the person's forecasted salary.

```
X_new = [[8,76492, 89]]
X_new = mm_scaler.transform(X_new)
X_new
```

```
array([[0.30434783, 0.34392983, 0.73170732]])
```

In this example, only a single line of values for the three features was processed. Of course, the number of rows of X_new can be a full data frame of new values, all processed at once.

Standardization Scaling

Another linear transformation of the data values converts the original data values to z-scores; this one taught in all introductory stat courses.

1. Standard score: The number of standard deviations a data value is from the mean.

Write the transformation of the data values for variable x as:

$$z_i = \frac{x_i - m}{s}$$

To standardize for each data value of a variable, for the i^{th} row of data, subtract the mean of the data and divide by the standard deviation of the data. The result: a distribution of z-scores, has a mean of 0 and a standard deviation of 1.

For this linear transformation, set $a = -(m/s)$ and $b = 1/s$, where s is the sample standard deviation and m is the sample mean.

The `StandardScaler` provides the computations for standardization of variables. If (not a requirement for standardization) a variable is normal, then most values will lie within 2.5 or 3 standard deviations from the mean, that is, standard scores of less than 2.5 and greater than -2.5.

```
from sklearn.preprocessing import StandardScaler
z_scaler = preprocessing.StandardScaler()
```

```
Xst = z_scaler.fit_transform(X)
Xst = pd.DataFrame(Xst, columns=['Years', 'Salary', 'Pre'])
Xst.head()
```

	Years	Salary	Pre
0	-0.443150	-0.509005	0.261780
1	0.995565	1.725666	1.407606
2	-0.443150	-0.771458	0.988865
3	-0.913988	-0.389457	1.759194
4	1.089581	1.170055	0.118991

The success of the transformation is shown by examining the mean and standard deviation of the three transformed, now standardized, variables.

```
round(Xst.mean(), 3)
```

```
Years     0.0
Salary    0.0
Pre       0.0
dtype: float64
```

```
round(Xst.std(), 3)
```

```
Years     1.018
Salary    1.018
Pre       1.018
dtype: float64
```

The range of the data roughly approximates that of normal data, though skewed right. In a perfectly normal distribution the standardized values would range from about -2.5 to 2.5.

```
Xst.min()
```

```
Years    -1.584917
Salary   -1.160138
Pre      -1.775039
dtype: float64
```

```
Xst.max()
```

```
Years     2.553042
Salary    2.827467
Pre       1.759194
dtype: float64
```

Can transform any data with the same values from `fit()`.

```
z_scaler.transform([[10, 115479.90, 90.0]])
```

```
array([[0.05065541, 2.72904472, 1.4074245 ]])
```

Can also transform manually. The computed values by which each data value is transformed in available from the computed `scale_` and `mean_` data structures. Each computed data structure contains one value for each variable to the data frame that is transformed.

```
print('mean:', z_scaler.mean_)
print('sd:', z_scaler.scale_)
```

```
mean: [9.21052632e+00 7.37962447e+04 8.26447368e+01]
sd:  [8.87458889e+00 2.26659629e+04 4.28169468e+00]
```

To illustrate, compute the transformed value of Salary for the second row of data in X.

```
(113278.04 - z_scaler.mean_[1]) / z_scaler.scale_[1]
```

```
1.7258672426031163
```

```
X.head()
```

Name	Years	Salary	Pre
Ritchie, Darnell	7.0	62788.56	83.0
Hoang, Binh	15.0	111078.48	88.0
Downs, Deborah	7.0	57139.90	85.0
Afshari, Anbar	3.0	58431.83	100.0
Knox, Michael	18.0	99052.98	81.0

Robust Scaling

Robust scaling resembles standardization, except it is more robust to the presence of outliers. The presence of outliers does not dramatically change the resulting scaled values as much as standardization in which an outlier can have a significant impact on the mean and an even bigger impact on increasing the size of the standard deviation (which depends on squared deviation scores).

Robust scaling accomplishes this robustness by replacing the mean in the standard score formula with the more robust median and the standard deviation with the more robust interquartile range. The median is the second quartile, and the IQR is the difference between the third and first quartiles. Unlike the mean and standard deviation, no matter how extreme a few values are in a distribution, the quartiles remain the same.

1. Robust scale score: The number of IQR's a data value is from the median.

Write the transformation of the data values for variable x as:

$$robust\ score = \frac{x_i - median}{IQR}$$

That is, to do a robust scaling, for each data value of a variable, for the i^{th} row of data subtract the median of the data and divide by the IQR of the data.

```
from sklearn.preprocessing import RobustScaler
r_scaler = preprocessing.RobustScaler()
```

```
Xrb = r_scaler.fit_transform(X)
Xrb = pd.DataFrame(Xrb, columns=['Years', 'Salary', 'Pre'])
Xrb.head()
```

	Years	Salary	Pre
0	-0.200	-0.244947	0.166166
1	0.700	1.502889	0.864865
2	-0.200	-0.408020	0.540541
3	-0.510	-0.083715	1.081081
4	1.100	1.001815	0.054054

The specific characteristics of the transformed variables differ from standardization but the general results remain. The means are somewhat close to 0. The standard deviations are less than 1 but certainly much closer to 1 than from the original distributions. The minimum and maximum values are less than the range of the standardized variables but roughly similar, again, especially compared to the original distributions.

```
round(Xrb.mean(), 3)
```

```
Years     0.0643
Salary    0.1467
Pre       0.0108
dtype: float64
```

```
round(Xrb.std(), 3)
```

```
Years     0.5185
Salary    0.7843
Pre       0.6061
dtype: float64
```

```
round(Xrb.min(), 3)
```

```
Years    -1.0000
Salary   -0.4619
Pre      -1.1351
dtype: float64
```

```
round(Xrb.max(), 3)
```

```
Years     1.7143
Salary    2.3692
Pre       1.0811
dtype: float64
```

5s completed at 10:11 PM