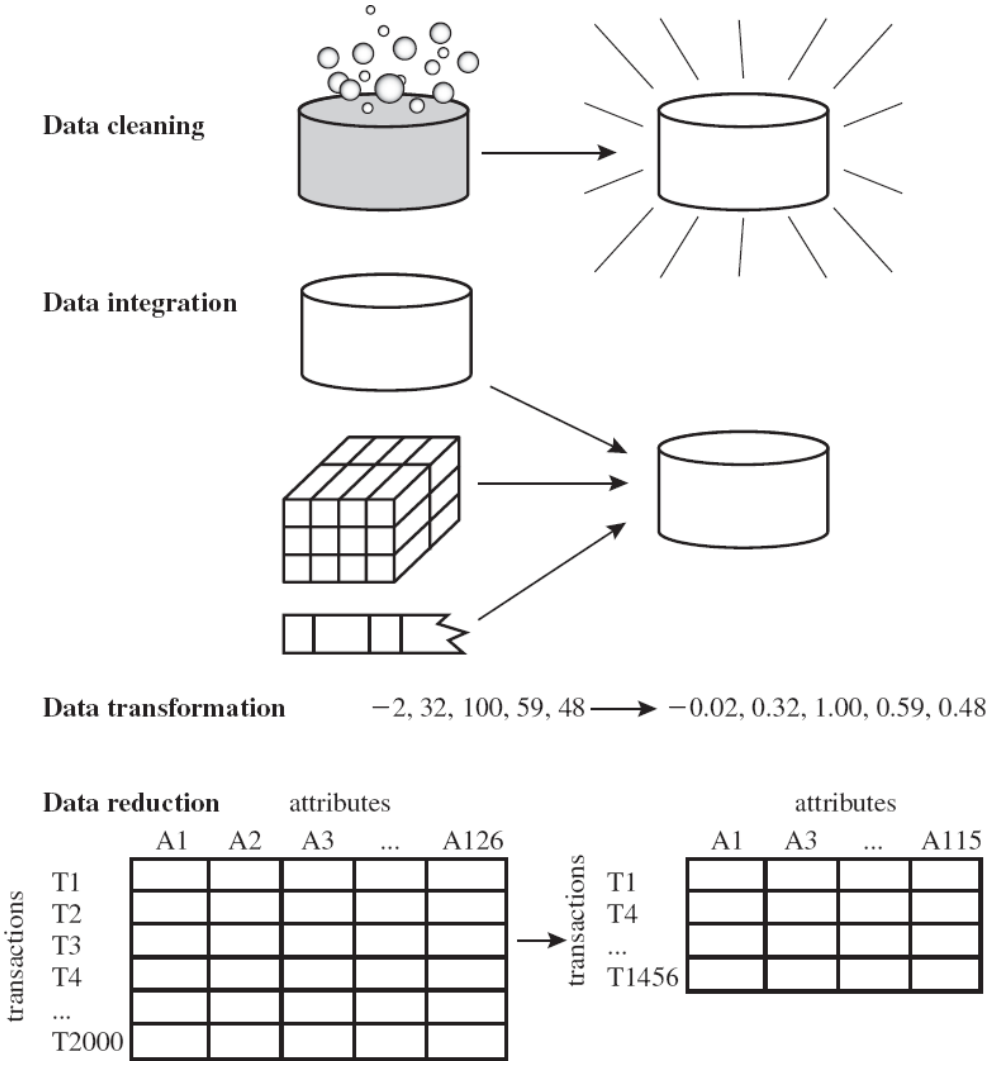
Lesson 6-A

Data Integration and Data Transformation

- Handling Redundancy, Summarizing, Aggregation, Binning, Normalization

Major Tasks in Data Preprocessing



Data Integration

- Combines data from multiple sources into a coherent store may cause *entity identification problem (schema conflict and value conflict)*
  - Schema conflict
    - *A.cust-id* ≡ *B.cust-number*
    - Integrate metadata from different sources
  - Data value conflict
    - “Bill Clinton” = “William Clinton”
    - Data codes for *pay_type* in one database may be “H” and “S” but “1” and “2” in another
    - Possible reasons: different representations, different scales, e.g., metric vs. British units

Handling Redundancy in Data Integration

- Redundant data occur often when integration of multiple databases
  - *Object identification*: The same attribute or object may have different names in different databases
  - *Derivable data:* One attribute may be a "derived" attribute in another table, e.g., age from dob
- Redundant attributes may be able to be detected by
  - Correlation coefficient for numeric data
  - Chi-square test for categorical data
- Careful integration of the data from multiple sources may help reduce/avoid redundancies and inconsistencies and improve mining speed and quality

4

Correlations between Attributes

- Correlation refers to the relationship between two attributes and how they related in terms of change.
  - Correlation analysis is used to detect attribute redundancy and improve performance of model
  - The performance of some machine learning models like linear and logistic regression may not be good if there are highly correlated attributes in the dataset.
  - We need to review all of the pairwise correlations of the attribute in the dataset.

Correlation Analysis

- Correlation analysis
  - Measure how strongly one attribute is related (dependent) with the other
  - Numeric data
    - Covariance
    - Pearson's Correlation Coefficient
    - `corr()` Pandas function
  - Categorical data
    - Chi-square test
    - `chi2_contingency()` SciPy function
    - `from sklearn.attribute_selection import chi2`

Pearson Correlation Coefficient between Attributes

$$corr(X,Y) = r_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y}$$

- $r_{X,Y}$ can take a range of values from +1 to -1
- If $r_{X,Y} > 0$: X and Y are positively correlated (X's values increase as Y's), The higher, the stronger correlation.
- $r_{X,Y} = 0$: independent;
- $r_{X,Y} < 0$: negatively correlated

Chi-Square Test

- Summarize the data in the two-way contingency table, the observed table
  - This table represents the observed counts of each cell
- Calculate the expected count for each cell in the table to find the expected table from observed counts
  - This table displays what the counts for each cell would be for sample data if there were not relations between two attributes
  - To find the expected count for each cell in the expected table we multiply the marginal row and column totals for that cell and divide by the overall total in the observed table
  - i.e. for each cell this is
  - $E = (row\ total \times column\ total) \,/\, total\ number\ of\ data$

Chi-Square Test

- Compute a Chi-Square test statistic as follows:
  - $\chi^2 = \Sigma(O_i - E_i)^2/E_i$
    - where O_i is an observed count of each cell of the contingency table, and E_i is an expected count of each cell of the contingency table
- With Chi-Square test statistic, a significance level (α) chosen, and the degree of freedom for the Chi-Square distribution, we can make a decision.
  - Degree of Freedom (df) = (number of rows – 1) x (numbers of columns – 1)
  - a significance level = 0.001

Chi-Square Test

- The decision is made by
  - Either comparing the value of test Chi-Square statistic to a critical chi-square value at a chosen significance level, α (rejection region approach)
    - Test statistic >= Critical value: reject null hypothesis, attributes are dependent (H_a)
    - Test statistic < Critical value: fail to reject null hypothesis, attributes are independent (H_0)
  - or finding the probability of getting of test Chi-Square statistic (p-value approach)
    - p-value <= α (a chosen significance level): reject null hypothesis, attributes are dependent (H_a)
    - p-value > α: fail to reject null hypothesis, attributes are independent (H_0)

Data Transformation

- We need to transform data in order that
  - resulting modeling process can be more efficient
  - modeling accuracy can be improved

How to Transform Data

- Summarizing and aggregation
  - Summarizing: summarizing (with group by features) using a statistical operation such as mean, max, min, standard deviation, etc. over dataset or specified feature(s)
  - Aggregation: aggregating (with group by features) using one or more statistical operations over the specified feature(s)
- Discretization
  - Raw values of a numeric feature are replaced by interval labels
  - Binning, histogram
- Normalization
  - Data are scaled so as to fall within a smaller range such as -1.0 to 1.0 or 0.0 to 1.0
  - z-score, min-max

Summarizing

- Summarizing the dataset or each feature
  - using different statistics such as `mean, max, min, sum, count, standard deviation`, etc.
- Summarizing groups in the dataset
  - `groupby` splits the dataset into different groups depending on a selected feature or more
  - Functions like `max, min, mean`, etc. can be applied to the `groupby` object

```
# import packages we need for loading the data set
import pandas as pd # to store tabular data
import numpy as np # to do some math
import matplotlib.pyplot as plt # a popular data visualization tool
import seaborn as sns # another popular data visualization tool

# allows the notebook to render graphics
%matplotlib inline

plt.style.use('fivethirtyeight') # a popular data visualization theme
```

```
# load in the data set. This data was scrapped from Basketball-reference
data = pd.read_csv("nba.csv") # load the CSV file using pandas.read_csv function
```

```
data.head()
```

|:

	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0

```
data.shape # (# rows, # cols)
```

|: (458, 9)

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 458 entries, 0 to 457
Data columns (total 9 columns):
Name        457 non-null object
Team        457 non-null object
Number      457 non-null float64
Position    457 non-null object
Age         457 non-null float64
Height      457 non-null object
Weight      457 non-null float64
College     373 non-null object
Salary      446 non-null float64
dtypes: float64(4), object(5)
memory usage: 32.3+ KB
```

```
data.describe()
```

6]:

	Number	Age	Weight	Salary
count	457.000000	457.000000	457.000000	4.460000e+02
mean	17.678337	26.938731	221.522976	4.842684e+06
std	15.966090	4.404016	26.368343	5.229238e+06
min	0.000000	19.000000	161.000000	3.088800e+04
25%	5.000000	24.000000	200.000000	1.044792e+06
50%	13.000000	26.000000	220.000000	2.839073e+06
75%	25.000000	30.000000	240.000000	6.500000e+06
max	99.000000	40.000000	307.000000	2.500000e+07

```
1 data['Team'].value_counts()
```

```
New Orleans Pelicans      19
Memphis Grizzlies         18
New York Knicks           16
Milwaukee Bucks           16
Denver Nuggets            15
Portland Trail Blazers    15
San Antonio Spurs         15
Brooklyn Nets             15
Toronto Raptors           15
Miami Heat                15
Sacramento Kings          15
Phoenix Suns              15
Atlanta Hawks             15
Washington Wizards        15
Utah Jazz                 15
Los Angeles Lakers        15
Chicago Bulls             15
Houston Rockets           15
Los Angeles Clippers      15
Philadelphia 76ers        15
Charlotte Hornets         15
Dallas Mavericks          15
Indiana Pacers            15
Golden State Warriors     15
Boston Celtics            15
Detroit Pistons           15
Oklahoma City Thunder     15
Cleveland Cavaliers       15
Minnesota Timberwolves    14
Orlando Magic             14
Name: Team, dtype: int64
```

```
1 data.groupby('Team').groups.keys()
```

```
dict_keys(['Atlanta Hawks', 'Boston Celtics', 'Brooklyn Nets', 'Charlotte Hornets', 'Chicago Bulls', 'Cleveland Cavaliers',
'Dallas Mavericks', 'Denver Nuggets', 'Detroit Pistons', 'Golden State Warriors', 'Houston Rockets', 'Indiana Pacers', 'Los
Angeles Clippers', 'Los Angeles Lakers', 'Memphis Grizzlies', 'Miami Heat', 'Milwaukee Bucks', 'Minnesota Timberwolves', 'Ne
w Orleans Pelicans', 'New York Knicks', 'Oklahoma City Thunder', 'Orlando Magic', 'Philadelphia 76ers', 'Phoenix Suns', 'Por
tland Trail Blazers', 'Sacramento Kings', 'San Antonio Spurs', 'Toronto Raptors', 'Utah Jazz', 'Washington Wizards'])
```

```
1 len(data.groupby('Team').groups['Los Angeles Lakers'])
```

```
15
```

```
1 data.groupby('Team').mean()
```

]:

Team	Number	Age	Weight	Salary
Atlanta Hawks	19.000000	28.200000	221.266667	4.860197e+06
Boston Celtics	31.866667	24.733333	219.466667	4.181505e+06
Brooklyn Nets	18.266667	25.600000	215.600000	3.501898e+06
Charlotte Hornets	17.133333	26.133333	220.400000	5.222728e+06
Chicago Bulls	19.200000	27.400000	218.933333	5.785559e+06
Cleveland Cavaliers	14.466667	29.533333	227.866667	7.642049e+06
Dallas Mavericks	20.000000	29.733333	227.000000	4.746582e+06
Denver Nuggets	15.266667	25.733333	217.533333	4.294424e+06
Detroit Pistons	17.266667	26.200000	222.200000	4.477884e+06
Golden State Warriors	20.866667	27.666667	224.600000	5.924600e+06
Houston Rockets	14.666667	26.866667	220.333333	5.018868e+06
Indiana Pacers	18.933333	26.400000	222.266667	4.450122e+06
Los Angeles Clippers	19.533333	29.466667	219.733333	6.323643e+06
Los Angeles Lakers	16.066667	27.533333	227.066667	4.784695e+06
Memphis Grizzlies	15.555556	28.388889	218.000000	5.467920e+06
Miami Heat	10.466667	28.933333	218.400000	6.347359e+06
Milwaukee Bucks	20.000000	24.562500	224.062500	4.350220e+06
Minnesota Timberwolves	19.571429	26.357143	228.642857	4.593054e+06
New Orleans Pelicans	17.000000	26.894737	221.000000	4.355304e+06
New York Knicks	13.250000	27.000000	223.625000	4.581494e+06
Oklahoma City Thunder	14.000000	27.066667	229.400000	6.251020e+06
Orlando Magic	16.428571	25.071429	213.357143	4.297248e+06
Philadelphia 76ers	18.066667	24.600000	222.133333	2.213778e+06
Phoenix Suns	15.466667	25.866667	218.600000	4.229676e+06
Portland Trail Blazers	16.000000	25.066667	218.600000	3.220121e+06
Sacramento Kings	16.933333	26.800000	221.333333	4.778911e+06
San Antonio Spurs	17.933333	31.600000	223.933333	5.629516e+06
Toronto Raptors	22.466667	26.133333	221.800000	4.741174e+06
Utah Jazz	17.866667	24.466667	220.000000	4.204006e+06
Washington Wizards	17.600000	27.866667	219.000000	5.088576e+06

```
1 data.groupby('Team')['Age', 'Weight'].mean()
```

]:

Team	Age	Weight
Atlanta Hawks	28.200000	221.266667
Boston Celtics	24.733333	219.466667
Brooklyn Nets	25.600000	215.600000
Charlotte Hornets	26.133333	220.400000
Chicago Bulls	27.400000	218.933333
Cleveland Cavaliers	29.533333	227.866667
Dallas Mavericks	29.733333	227.000000
Denver Nuggets	25.733333	217.533333
Detroit Pistons	26.200000	222.200000
Golden State Warriors	27.666667	224.600000
Houston Rockets	26.866667	220.333333
Indiana Pacers	26.400000	222.266667
Los Angeles Clippers	29.466667	219.733333
Los Angeles Lakers	27.533333	227.066667
Memphis Grizzlies	28.388889	218.000000
Miami Heat	28.933333	218.400000
Milwaukee Bucks	24.562500	224.062500
Minnesota Timberwolves	26.357143	228.642857
New Orleans Pelicans	26.894737	221.000000
New York Knicks	27.000000	223.625000
Oklahoma City Thunder	27.066667	229.400000
Orlando Magic	25.071429	213.357143
Philadelphia 76ers	24.600000	222.133333
Phoenix Suns	25.866667	218.600000
Portland Trail Blazers	25.066667	218.600000
Sacramento Kings	26.800000	221.333333
San Antonio Spurs	31.600000	223.933333
Toronto Raptors	26.133333	221.800000
Utah Jazz	24.466667	220.000000
Washington Wizards	27.866667	219.000000

Aggregation

- Aggregate using one or more operations over the specified features per group
  - `pandas.DataFrame.aggregate`
  - *`agg`* is an alias for *`aggregate`*. Use the alias.

```
1 data.groupby(['Team', 'Position'])['Age', 'Weight'].mean()
```

Team	Position	Age	Weight
Atlanta Hawks	C	28.333333	250.00
	PF	28.250000	239.50
	PG	24.500000	179.00
	SF	29.000000	210.50
	SG	29.500000	208.00
...	...	...	...
Washington Wizards	C	30.666667	244.00
	PF	30.000000	247.50
	PG	27.500000	192.50
	SF	25.500000	208.25
	SG	27.250000	210.00

149 rows × 2 columns

```
1 data.groupby(['Team', 'Position']).agg({'Age': 'mean', 'Weight':'mean'})
```

Team	Position	Age	Weight
Atlanta Hawks	C	28.333333	250.00
	PF	28.250000	239.50
	PG	24.500000	179.00
	SF	29.000000	210.50
	SG	29.500000	208.00
...	...	...	...
Washington Wizards	C	30.666667	244.00
	PF	30.000000	247.50
	PG	27.500000	192.50
	SF	25.500000	208.25
	SG	27.250000	210.00

149 rows × 2 columns

```
1 data.groupby(['Team', 'Position']).agg({'Age': ['mean', 'max', 'min'], 'Weight':['mean', 'max', 'min']})
```

2]:

		Age			Weight		
		mean	max	min	mean	max	min
Team	Position						
Atlanta Hawks	C	28.333333	31.0	24.0	250.00	260.0	245.0
	PF	28.250000	31.0	24.0	239.50	246.0	235.0
	PG	24.500000	27.0	22.0	179.00	186.0	172.0
	SF	29.000000	32.0	26.0	210.50	220.0	201.0
	SG	29.500000	35.0	24.0	208.00	225.0	190.0
...	...	...	...	...	...	...	...
Washington Wizards	C	30.666667	33.0	27.0	244.00	250.0	240.0
	PF	30.000000	34.0	26.0	247.50	250.0	245.0
	PG	27.500000	30.0	25.0	192.50	195.0	190.0
	SF	25.500000	30.0	20.0	208.25	225.0	198.0
	SG	27.250000	33.0	22.0	210.00	220.0	195.0

149 rows × 6 columns

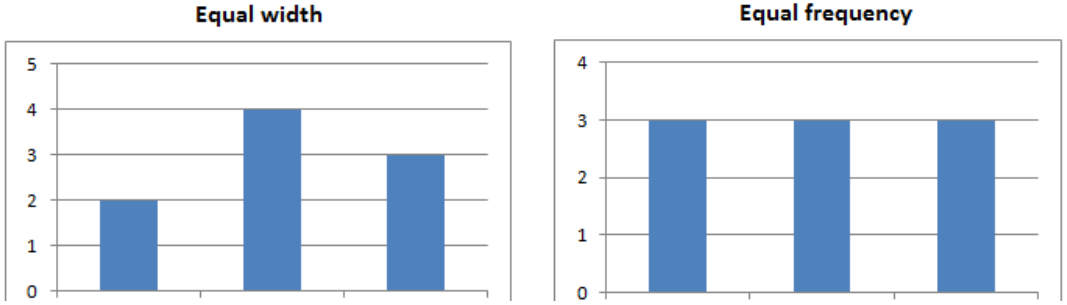
```
1 data.groupby('Team').agg({'Age': ['mean', 'max', 'min'], 'Weight':['mean', 'max', 'min']})
```

]:

	Age			Weight		
	mean	max	min	mean	max	min
Team						
Atlanta Hawks	28.200000	35.0	22.0	221.266667	260.0	172.0
Boston Celtics	24.733333	29.0	20.0	219.466667	260.0	180.0
Brooklyn Nets	25.600000	32.0	21.0	215.600000	275.0	175.0
Charlotte Hornets	26.133333	31.0	21.0	220.400000	289.0	184.0
Chicago Bulls	27.400000	35.0	21.0	218.933333	275.0	161.0
Cleveland Cavaliers	29.533333	35.0	24.0	227.866667	275.0	179.0
Dallas Mavericks	29.733333	37.0	22.0	227.000000	275.0	185.0
Denver Nuggets	25.733333	36.0	20.0	217.533333	280.0	175.0
Detroit Pistons	26.200000	36.0	20.0	222.200000	279.0	172.0
Golden State Warriors	27.666667	33.0	20.0	224.600000	273.0	175.0
Houston Rockets	26.866667	38.0	22.0	220.333333	265.0	185.0
Indiana Pacers	26.400000	30.0	20.0	222.266667	255.0	180.0
Los Angeles Clippers	29.466667	39.0	23.0	219.733333	265.0	175.0
Los Angeles Lakers	27.533333	37.0	20.0	227.066667	270.0	175.0
Memphis Grizzlies	28.388889	39.0	21.0	218.000000	270.0	165.0
Miami Heat	28.933333	36.0	20.0	218.400000	265.0	165.0
Milwaukee Bucks	24.562500	32.0	19.0	224.062500	265.0	190.0
Minnesota Timberwolves	26.357143	40.0	20.0	228.642857	307.0	189.0
New Orleans Pelicans	26.894737	31.0	23.0	221.000000	270.0	170.0
New York Knicks	27.000000	34.0	20.0	223.625000	278.0	195.0
Oklahoma City Thunder	27.066667	38.0	21.0	229.400000	255.0	185.0
Orlando Magic	25.071429	32.0	20.0	213.357143	260.0	169.0
Philadelphia 76ers	24.600000	37.0	20.0	222.133333	275.0	175.0
Phoenix Suns	25.866667	33.0	19.0	218.600000	260.0	175.0
Portland Trail Blazers	25.066667	34.0	20.0	218.600000	265.0	173.0
Sacramento Kings	26.800000	36.0	22.0	221.333333	270.0	175.0
San Antonio Spurs	31.600000	40.0	22.0	223.933333	290.0	185.0
Toronto Raptors	26.133333	36.0	20.0	221.800000	255.0	190.0
Utah Jazz	24.466667	28.0	20.0	220.000000	265.0	179.0
Washington Wizards	27.866667	34.0	20.0	219.000000	250.0	190.0

Binning

- Binning methods sort the data in ascending order and then partition them into a set of equal-frequency or equal-size bins.
  - Equal-width (distance) binning
  - Equal-frequency (depth) binning
- There are several different terms for binning
  - bucketing, discrete binning, discretization or quantization



Equal-frequency binning

- It divides the range into N intervals, each containing the same number of samples
  - Data scaling (smooth out the data to remove the noise)
    - Smoothing by bin means
    - Smoothing by bin medians
    - Smoothing by bin boundaries
- Given the sorted data:
  - 0, 4, 12, 16, 16, 18, 24, 26, 28
  - Equal-frequency bins of size 3 (i.e. each bin contains three values)

Equal-width (distance) binning

- It divides the range into N intervals of equal size:
  - If A and B are the lowest and highest values of the feature, the width of intervals with be: $W = (B - A)/ N$.
- Given the sorted data:
  - 0, 4, 12, 16, 16, 18, 24, 26, 28
- Equal-width bins of size 10 (i.e. each bin's interval range is 10)

pandas.qcut

- `qcut` is a quantile-based discretization function.
  - divides up the underlying data into equal size bins to make sure the **distribution** of data in the bins is (almost) equal
  - defines the bins using percentiles based on the distribution of the data
  - Using `qcut`, we can create 4 bins (aka quartiles), 5 bins (aka quintiles) and 10 bins (aka deciles)

```
1 data['Glucose'].describe()
```

```
count    768.000000
mean     120.894531
std       31.972618
min        0.000000
25%       99.000000
50%      117.000000
75%      140.250000
max      199.000000
Name: Glucose, dtype: float64
```

```
1 pd.qcut(data['Glucose'], q=4)
```

```
0      (140.25, 199.0]
1       (-0.001, 99.0]
2      (140.25, 199.0]
3       (-0.001, 99.0]
4      (117.0, 140.25]
             ...
763      (99.0, 117.0]
764    (117.0, 140.25]
765    (117.0, 140.25]
766    (117.0, 140.25]
767     (-0.001, 99.0]
Name: Glucose, Length: 768, dtype: category
Categories (4, interval[float64]): [(-0.001, 99.0] < (99.0, 117.0] < (117.0, 140.25] < (140.25, 199.0]]
```

```
1 pd.qcut(data['Glucose'], q=4).value_counts()
```

```
(-0.001, 99.0]     197
(99.0, 117.0]      194
(140.25, 199.0]    192
(117.0, 140.25]    185
Name: Glucose, dtype: int64
```

```
1 pd.qcut(data['Glucose'], q=10)
```

```
79]: 0      (147.0, 167.0]
     1      (43.999, 86.2]
     2      (167.0, 199.0]
     3        (86.2, 95.0]
     4      (135.0, 147.0]
                 ...
     763     (95.0, 102.0]
     764    (117.0, 125.0]
     765    (117.0, 125.0]
     766    (125.0, 135.0]
     767      (86.2, 95.0]
     Name: Glucose, Length: 768, dtype: category
     Categories (10, interval[float64]): [(43.999, 86.2] < (86.2, 95.0] < (95.0, 102.0] < (102.0, 109.0] ... (125.0, 135.0] < (13
     5.0, 147.0] < (147.0, 167.0] < (167.0, 199.0]]
```

```
1 pd.qcut(data['Glucose'], q=10).value_counts()
```

```
30]: (117.0, 125.0]    80
     (135.0, 147.0]    78
     (102.0, 109.0]    78
     (86.2, 95.0]      78
     (109.0, 117.0]    77
     (43.999, 86.2]    77
     (167.0, 199.0]    76
     (95.0, 102.0]     76
     (147.0, 167.0]    72
     (125.0, 135.0]    71
     Name: Glucose, dtype: int64
```

In [98]:
```
bin_labels_5 = ['very low', 'low', 'medium', 'high', 'very high']
data['quantile_glucose'] = pd.qcut(data['Glucose'], q = 5, labels=bin_labels_5)
data.head()
```

Out[98]:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome	quantile_glucose
0	6	148.0	72.0	35.0	NaN	33.6	0.627	50	1	very high
1	1	85.0	66.0	29.0	NaN	26.6	0.351	31	0	very low
2	8	183.0	64.0	NaN	NaN	23.3	0.672	32	1	very high
3	1	89.0	66.0	23.0	94.0	28.1	0.167	21	0	very low
4	0	137.0	40.0	35.0	168.0	43.1	2.288	33	1	high

In [99]:
```
data['quantile_glucose'].value_counts()
```

Out[99]:
```
medium       157
very low     155
low          154
high         149
very high    148
Name: quantile_glucose, dtype: int64
```

pandas.cut

- Bin values into discrete intervals
  - Use *cut* when you need to segment and sort data values into equal width bins
    - No guarantee about the distribution of values in each bin
  - This function is also useful for going from a continuous variable to a categorical variable
    - convert ages to groups of age ranges

```
1 pd.cut(data['Glucose'], bins=4)
```

```
: 0       (99.5, 149.25]
  1        (49.75, 99.5]
  2      (149.25, 199.0]
  3        (49.75, 99.5]
  4       (99.5, 149.25]
             ...
  763     (99.5, 149.25]
  764     (99.5, 149.25]
  765     (99.5, 149.25]
  766     (99.5, 149.25]
  767      (49.75, 99.5]
  Name: Glucose, Length: 768, dtype: category
  Categories (4, interval[float64]): [(-0.199, 49.75] < (49.75, 99.5] < (99.5, 149.25] < (149.25, 199.0]]
```

```
1 pd.cut(data['Glucose'], bins=4).value_counts()
```

```
: (99.5, 149.25]     428
  (49.75, 99.5]      191
  (149.25, 199.0]    143
  (-0.199, 49.75]      6
  Name: Glucose, dtype: int64
```

qcut vs cut

- If you want equal distribution of the values in the bins, use `qcut`
- If you want to define your own numeric bin ranges (bin edges), then use `cut`

```
1 data['Glucose'].describe()
```

```
count    768.000000
mean     120.894531
std       31.972618
min        0.000000
25%       99.000000
50%      117.000000
75%      140.250000
max      199.000000
Name: Glucose, dtype: float64
```

```
1 pd.qcut(data['Glucose'], q=4)
```

```
0      (140.25, 199.0]
1       (-0.001, 99.0]
2      (140.25, 199.0]
3       (-0.001, 99.0]
4      (117.0, 140.25]
             ...
763     (99.0, 117.0]
764    (117.0, 140.25]
765    (117.0, 140.25]
766    (117.0, 140.25]
767     (-0.001, 99.0]
Name: Glucose, Length: 768, dtype: category
Categories (4, interval[float64]): [(-0.001, 99.0] < (99.0, 117.0] < (117.0, 140.25] < (140.25, 199.0]]
```

```
1 pd.qcut(data['Glucose'], q=4).value_counts()
```

```
(-0.001, 99.0]     197
(99.0, 117.0]      194
(140.25, 199.0]    192
(117.0, 140.25]    185
Name: Glucose, dtype: int64
```

```
1 pd.cut(data['Glucose'], bins=4)
```

```
0      (99.5, 149.25]
1       (49.75, 99.5]
2     (149.25, 199.0]
3       (49.75, 99.5]
4      (99.5, 149.25]
             ...
763    (99.5, 149.25]
764    (99.5, 149.25]
765    (99.5, 149.25]
766    (99.5, 149.25]
767     (49.75, 99.5]
Name: Glucose, Length: 768, dtype: category
Categories (4, interval[float64]): [(-0.199, 49.75] < (49.75, 99.5] < (99.5, 149.25] < (149.25, 199.0]]
```

```
1 pd.cut(data['Glucose'], bins=4).value_counts()
```

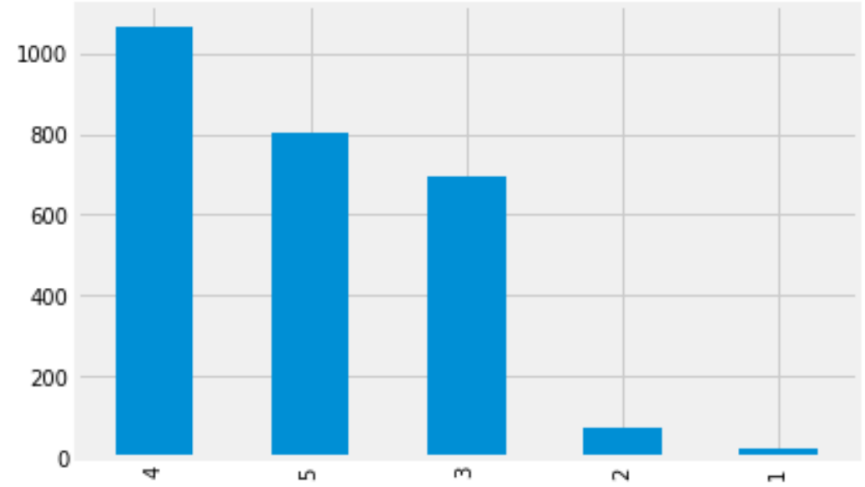
```
(99.5, 149.25]     428
(49.75, 99.5]      191
(149.25, 199.0]    143
(-0.199, 49.75]      6
Name: Glucose, dtype: int64
```

Bar Plot and Histogram

- If a feature is categorical, a bar plot is preferred
  - The height of the bar the indicates the frequency (count) of the feature.
  - This is called a bar graph
- If a feature is numeric, a histogram is preferred.
  - The range of values for a feature is partitioned into disjoin consecutive subranges.
  - The subrange referred as buckets or bins are disjoint subset of the data consecutive for a feature
  - The range of a bucket is known a width
  - This is called a histogram

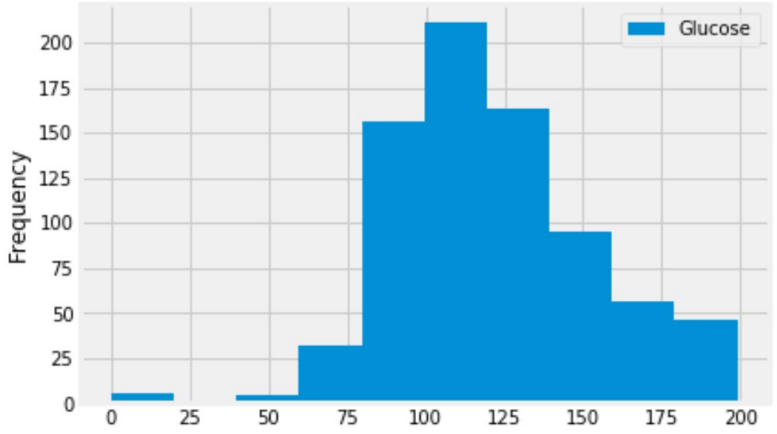
```
1 art_ratings.value_counts().plot(kind='bar')
```

```
: <matplotlib.axes._subplots.AxesSubplot at 0x1fa70b29208>
```




```
1 # get a histogram of the Glucose column for diabetes class
2 data.plot('Outcome', 'Glucose', kind = 'hist')
```

```
9]: <matplotlib.axes._subplots.AxesSubplot at 0x1da8b1f8c48>
```



Normalization

- Normalizing the data attempts to give all attributes an equal weight
  - The measurement unit used can affect the modeling and data analysis
  - To avoid dependence on the choice of measurement units, the data should be *normalized* or *standardized*
    - This involves transforming the data to fall within a smaller or common range such as [-1, 1] or [0.0, 1.0].
  - Normalization is particularly useful for classification

```
1 data_imputed_mean = pd.DataFrame(data_imputed, columns=data_column_names)
```

```
1 data_imputed_mean.head()
```

:

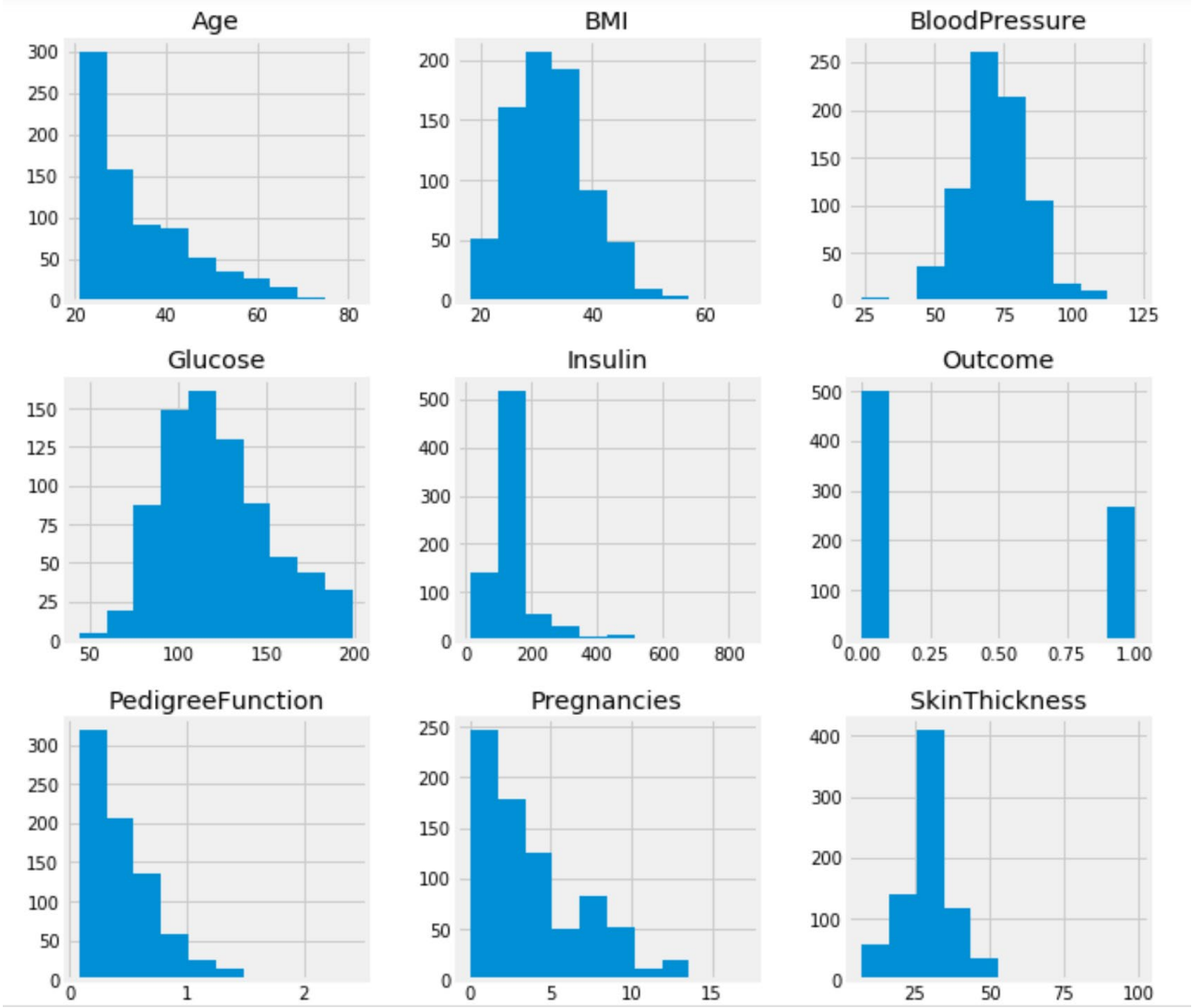
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	PedigreeFunction	Age	Outcome
0	6.0	148.0	72.0	35.00000	155.548223	33.6	0.627	50.0	1.0
1	1.0	85.0	66.0	29.00000	155.548223	26.6	0.351	31.0	0.0
2	8.0	183.0	64.0	29.15342	155.548223	23.3	0.672	32.0	1.0
3	1.0	89.0	66.0	23.00000	94.000000	28.1	0.167	21.0	0.0
4	0.0	137.0	40.0	35.00000	168.000000	43.1	2.288	33.0	1.0

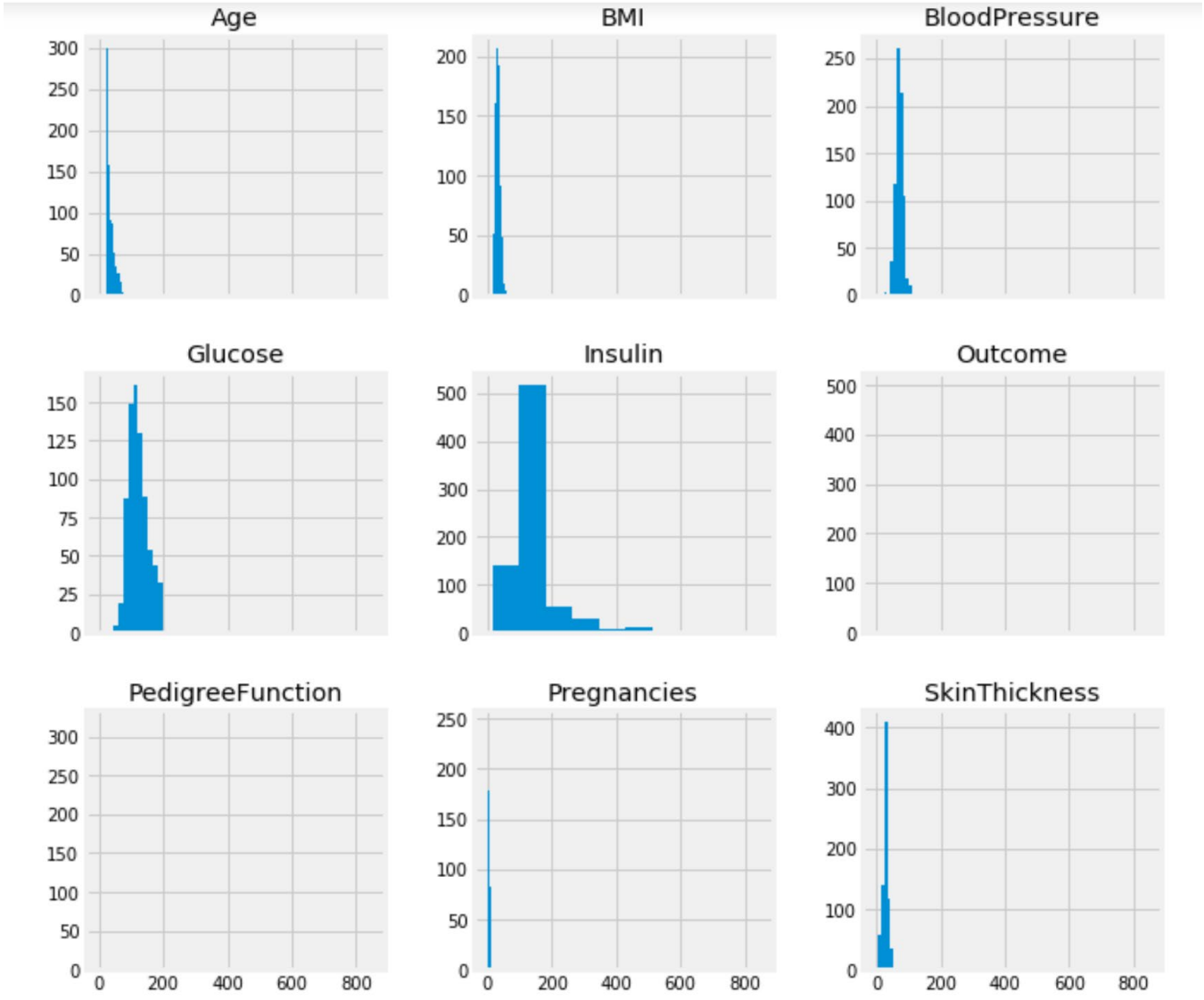
```
1 data_imputed_mean.describe()
```

:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	PedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	121.686763	72.405184	29.153420	155.548223	32.457464	0.471876	33.240885	0.348958
std	3.369578	30.435949	12.096346	8.790942	85.021108	6.875151	0.331329	11.760232	0.476951
min	0.000000	44.000000	24.000000	7.000000	14.000000	18.200000	0.078000	21.000000	0.000000
25%	1.000000	99.750000	64.000000	25.000000	121.500000	27.500000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.202592	29.153420	155.548223	32.400000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	155.548223	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

```
1 data_imputed_mean.hist(figsize=(10,10))
```


Age
BMI
BloodPressure
Glucose
Insulin
Outcome
PedigreeFunction
Pregnancies
SkinThickness



It is clear that each column uses a different scale weight.
Normalization ensures that all values are treated equally for the model.

Z-score Normalization

- The process of converting a raw data into a z-score (standard) score is called z—score **standardizing** or **normalizing**

$$v' = \frac{v - \mu_A}{\sigma_A}$$

  - v: the value of a feature A,
  - v': the normalized value,
  - μ_A: the mean of the feature A,
  - σ_A: the standard deviation of the feature A

  - Ex. Let μ_A = 54,000, σ_A = 16,000. Then the normalized value of 73,600 is

Z-score

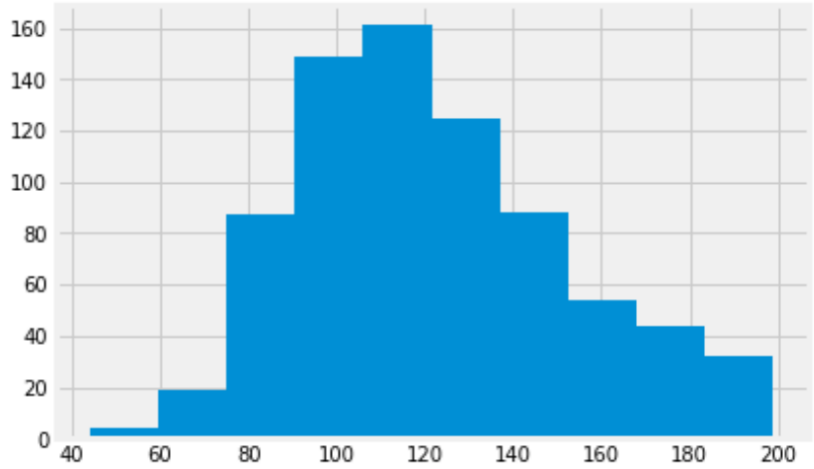
- A **Z-score** is a numerical measurement used in statistics of a **value's** relationship to the mean (average) of a group of values, measured in terms of standard deviations from the mean.
  - If a Z-score is 0, it indicates that the data point is identical to the mean
  - If a Z-scores is positive, it indicates that the score is above the mean
    - A Z-score of 1.0 would indicate a value that is one standard deviation from the mean
  - If a Z-scores is negative, it indicates that the score is below the mean
- $z = \frac{x-\mu}{\sigma}$

```
1  data['Glucose'].head()
```

```
35]: 0    148.0
     1     85.0
     2    183.0
     3     89.0
     4    137.0
     Name: Glucose, dtype: float64
```

```
1  data['Glucose'].hist()
```

```
34]: <matplotlib.axes._subplots.AxesSubplot at 0x176daf5ba88>
```




```
1  mean = data['Glucose'].mean()
```

```
1  std = data['Glucose'].std()
```

```
1  data['Glucose'].mean(), data['Glucose'].std()
```

```
38]: (121.6867627785059, 30.53564107280403)
```

```
1 z_score = (data['Glucose']-mean)/std
```

```
1 z_score.head()
```

```
0    0.861722
1   -1.201441
2    2.007924
3   -1.070446
4    0.501487
Name: Glucose, dtype: float64
```

```
1 z_score.mean(), z_score.std()
```

```
(1.0025473970730444e-16, 1.000000000000001)
```

```
1 print("%.3f" % z_score.mean())
```

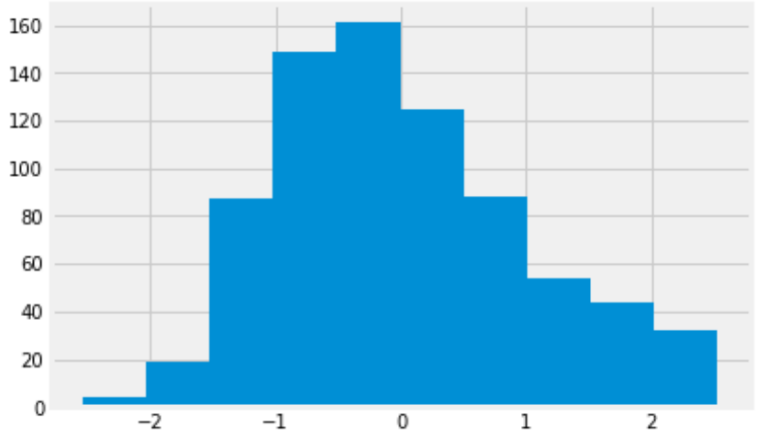
```
0.000
```

```
1 print("%.3f" % z_score.std())
```

```
1.000
```

```
1 z_score.hist()
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x176daffe1c8>
```



```
1  # built-in z-score normalizer
```

```
1  from sklearn.preprocessing import StandardScaler
```

```
1  scaler = StandardScaler ()
```

```
1  z_score_glucose = scaler.fit_transform(data[['Glucose']])
```

```
1  type (z_score_glucose)
```

8]: numpy.ndarray

```
1  z_score_glucose_df = pd.DataFrame(z_score_glucose, columns=['Glucose'])
```

```
1  z_score_glucose_df.head()
```

6]:

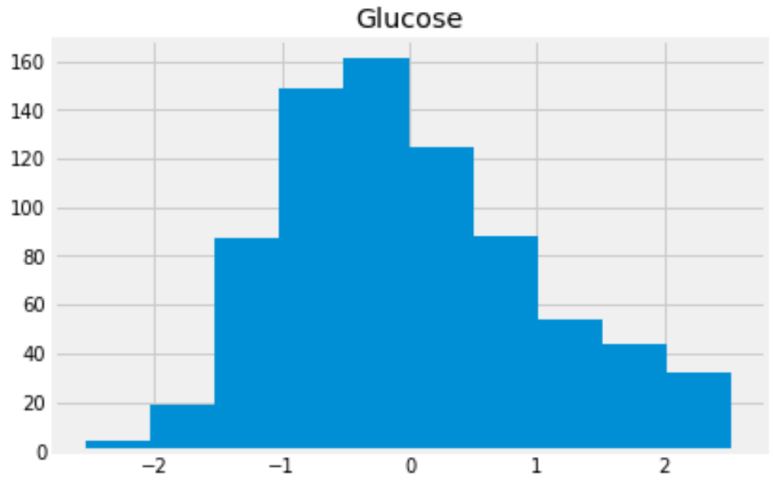
	Glucose
0	0.862287
1	-1.202229
2	2.009241
3	-1.071148
4	0.501816

```
1 z_score_glucose_df.mean(), z_score_glucose_df.std()
```

```
: (Glucose    1.316844e-16
   dtype: float64, Glucose    1.000656
   dtype: float64)
```

```
1 z_score_glucose_df.hist()
```

```
: array([[<matplotlib.axes._subplots.AxesSubplot object at 0x000001C568558588>]],
      dtype=object)
```



```
1 data_imputed_mean.describe()
```

:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	PedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	121.686763	72.405184	29.153420	155.548223	32.457464	0.471876	33.240885	0.348958
std	3.369578	30.435949	12.096346	8.790942	85.021108	6.875151	0.331329	11.760232	0.476951
min	0.000000	44.000000	24.000000	7.000000	14.000000	18.200000	0.078000	21.000000	0.000000
25%	1.000000	99.750000	64.000000	25.000000	121.500000	27.500000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.202592	29.153420	155.548223	32.400000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	155.548223	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

```
column_names = ['Pregnancies', 'Glucose', 'BloodPressure', 'SkinThickness', 'Insulin', 'BMI',
                'DiabetesPedigreeFunction', 'Age', 'Outcome']
```

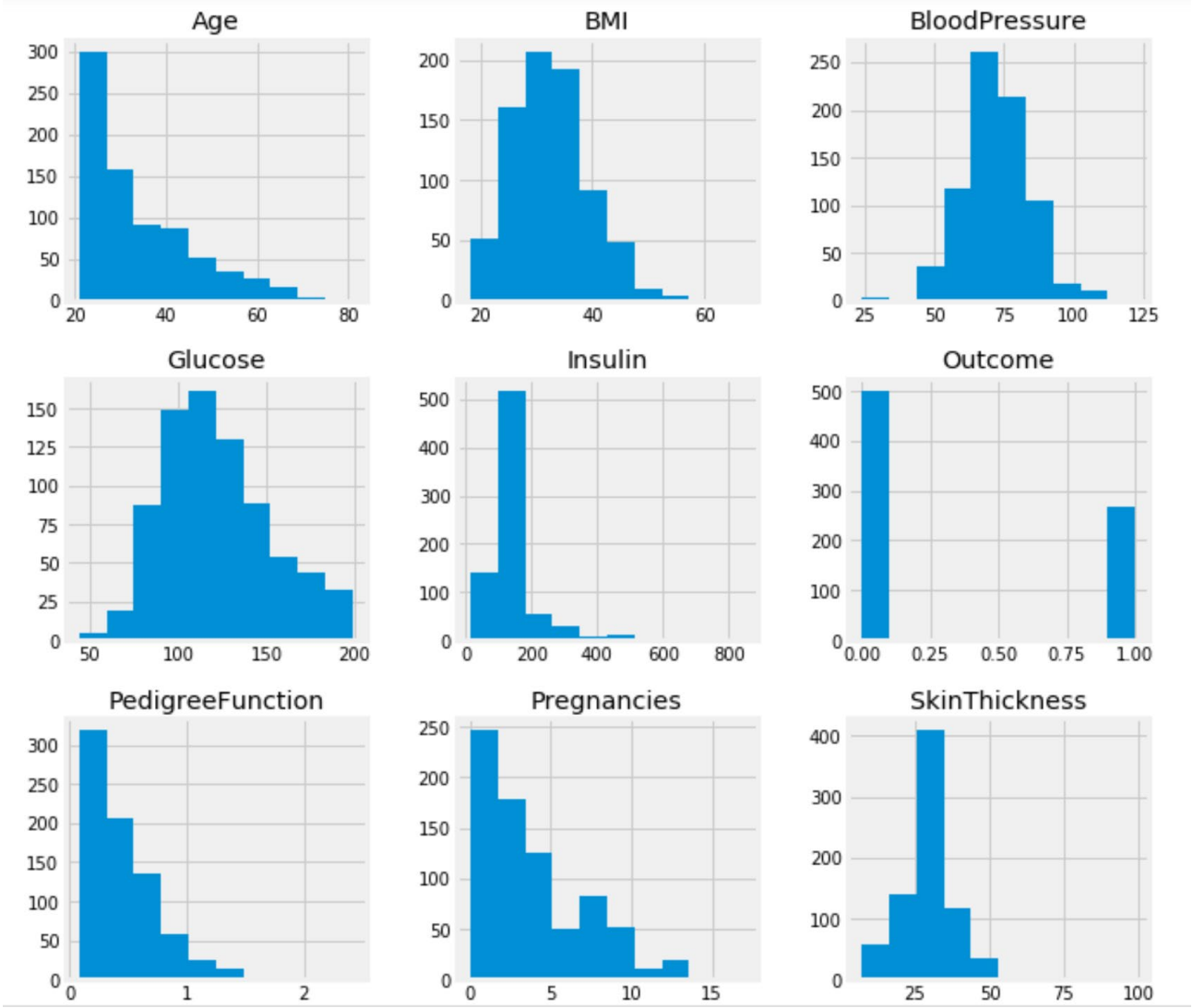
```
data_imputed_mean_z_score_scaled = pd.DataFrame(scaler.fit_transform(data_imputed_mean), columns = data_column_names)
```

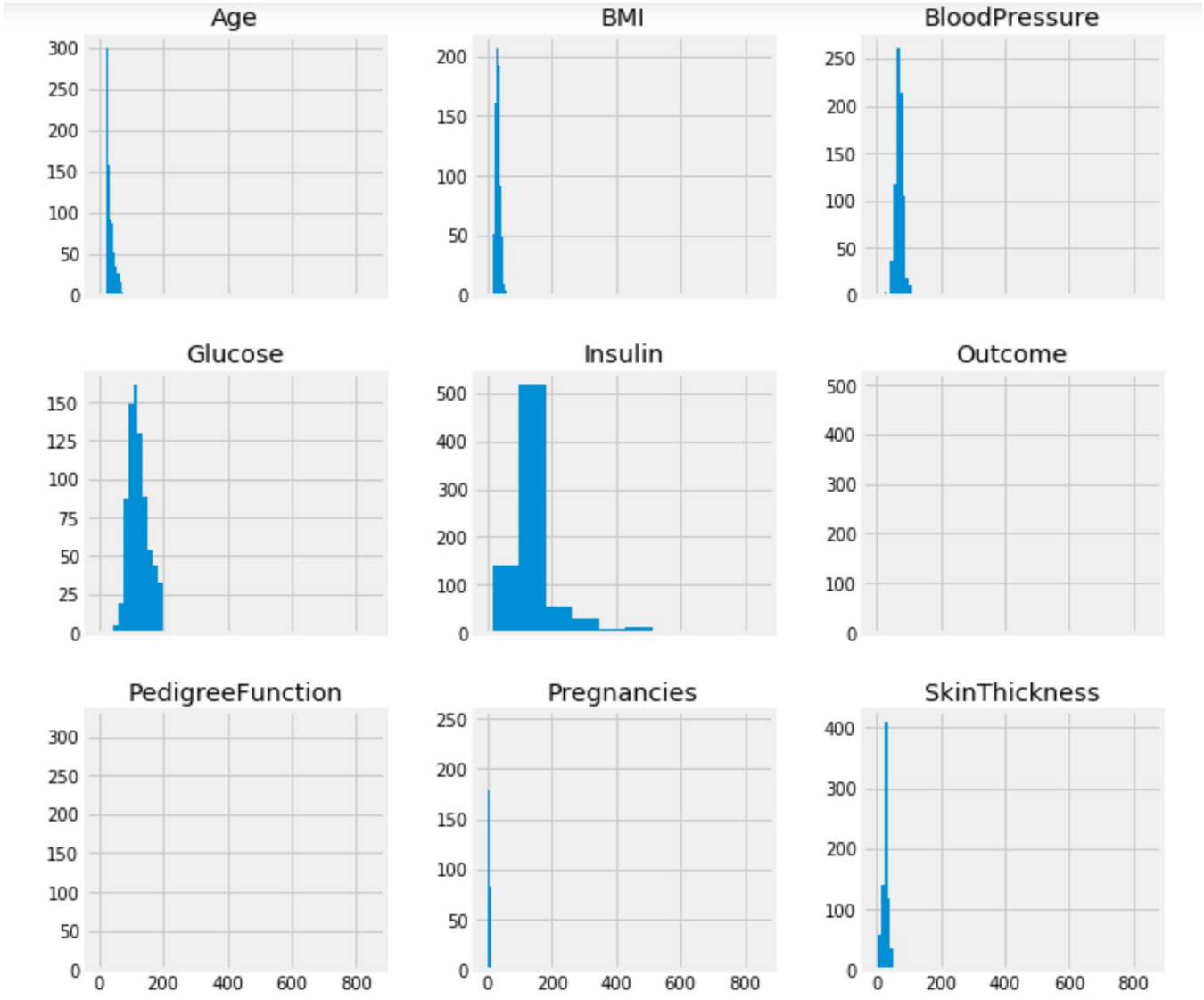
```
data_imputed_mean_z_score_scaled.describe()
```

```
1 data_imputed_mean_z_score_scaled.describe()
```

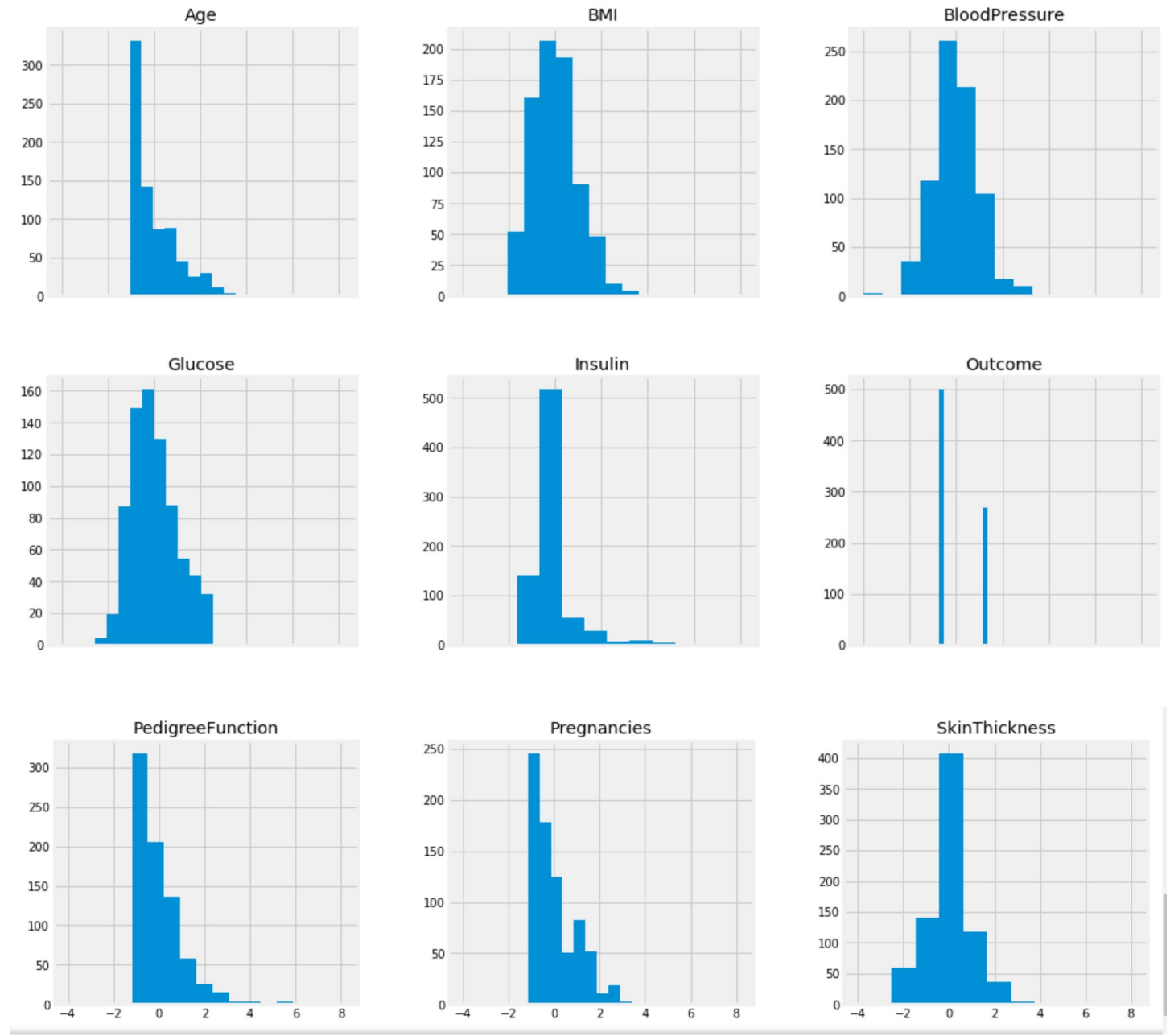
:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	PedigreeFunction	Age	Outcome
count	7.680000e+02	7.680000e+02	7.680000e+02	7.680000e+02	7.680000e+02	7.680000e+02	7.680000e+02	7.680000e+02	7.680000e+02
mean	2.544261e-17	-3.301757e-16	6.966722e-16	6.866252e-16	-2.352033e-16	3.553292e-16	2.398978e-16	1.857600e-16	2.408374e-16
std	1.000652e+00	1.000652e+00	1.000652e+00	1.000652e+00	1.000652e+00	1.000652e+00	1.000652e+00	1.000652e+00	1.000652e+00
min	-1.141852e+00	-2.554131e+00	-4.004245e+00	-2.521670e+00	-1.665945e+00	-2.075119e+00	-1.189553e+00	-1.041549e+00	-7.321202e-01
25%	-8.448851e-01	-7.212214e-01	-6.953060e-01	-4.727737e-01	-4.007289e-01	-7.215397e-01	-6.889685e-01	-7.862862e-01	-7.321202e-01
50%	-2.509521e-01	-1.540881e-01	-1.675912e-02	8.087936e-16	-3.345079e-16	-8.363615e-03	-3.001282e-01	-3.608474e-01	-7.321202e-01
75%	6.399473e-01	6.103090e-01	6.282695e-01	3.240194e-01	-3.345079e-16	6.029301e-01	4.662269e-01	6.602056e-01	1.365896e+00
max	3.906578e+00	2.541850e+00	4.102655e+00	7.950467e+00	8.126238e+00	5.042087e+00	5.883565e+00	4.063716e+00	1.365896e+00


Age
BMI
BloodPressure
Glucose
Insulin
Outcome
PedigreeFunction
Pregnancies
SkinThickness



It is clear that each column uses a different scale weight.
Normalization ensures that all values are treated equally for the model.


Age
BMI
BloodPressure
Glucose
Insulin
Outcome
PedigreeFunction
Pregnancies
SkinThickness

Min-Max Normalization

- Performs a linear transformation on the raw data based on the minimum and maximum value.
  - Suppose that min_A and max_A are the minimum and maximum values of a feature, *A*.
  - Min-max normalization maps a value, v_i , of *A* to v_i' in the range [*new* min_A, *new* max_A] by computing

$$v' = \frac{v - min_A}{max_A - min_A}(new\_max_A - new\_min_A) + new\_min_A$$

  - Let income range \$12,000 to \$98,000 normalized to [0.0, 1.0]. Then \$73,000 is mapped to ?

```
# import the sklearn module
from sklearn.preprocessing import MinMaxScaler
```

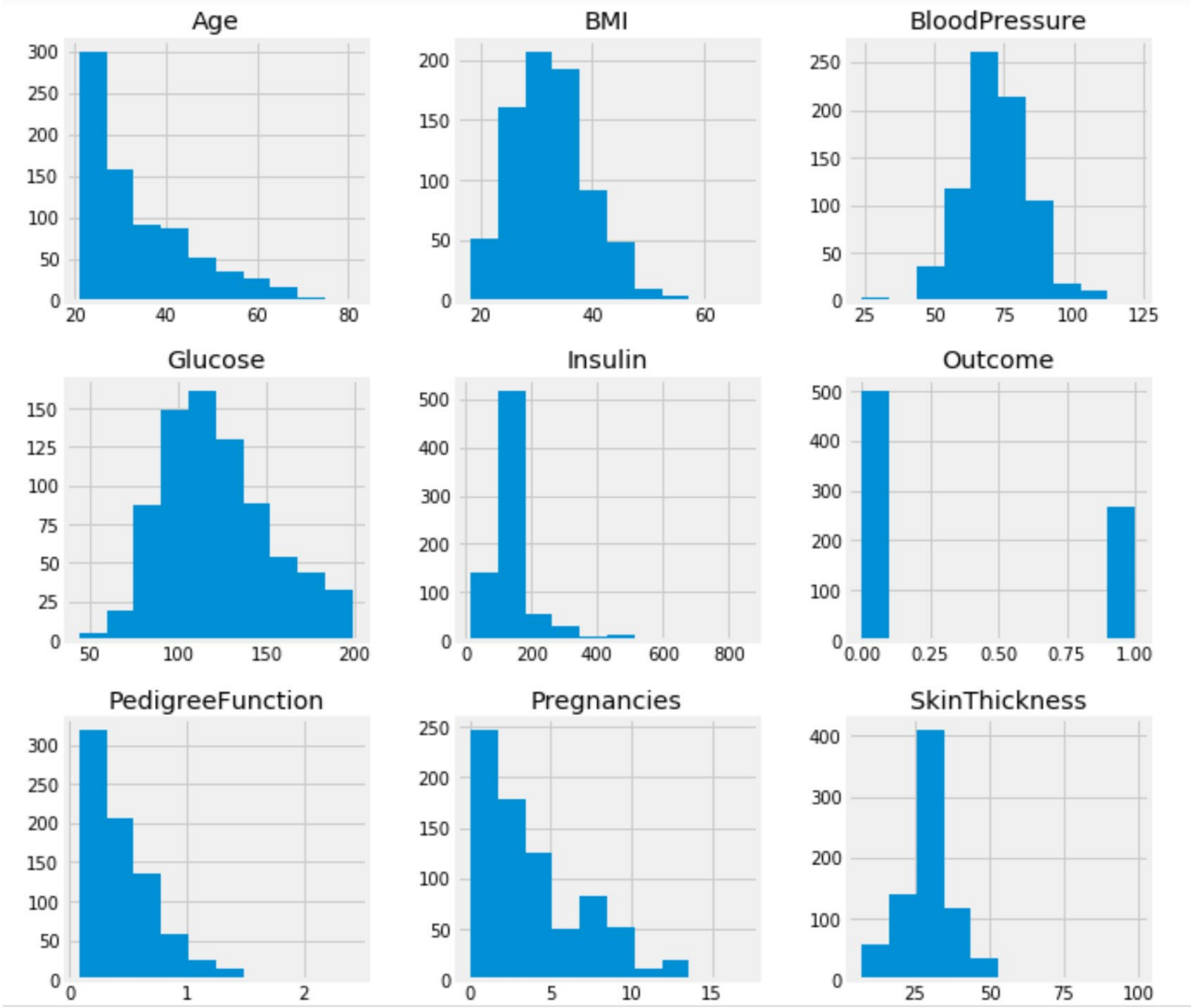
```
# instantiate the class
min_max = MinMaxScaler()

# apply the min_min normalization (scaling)
data_imputed_minmax_scaled = pd.DataFrame(min_max.fit_transform(data_imputed), columns=data_column_names)
```

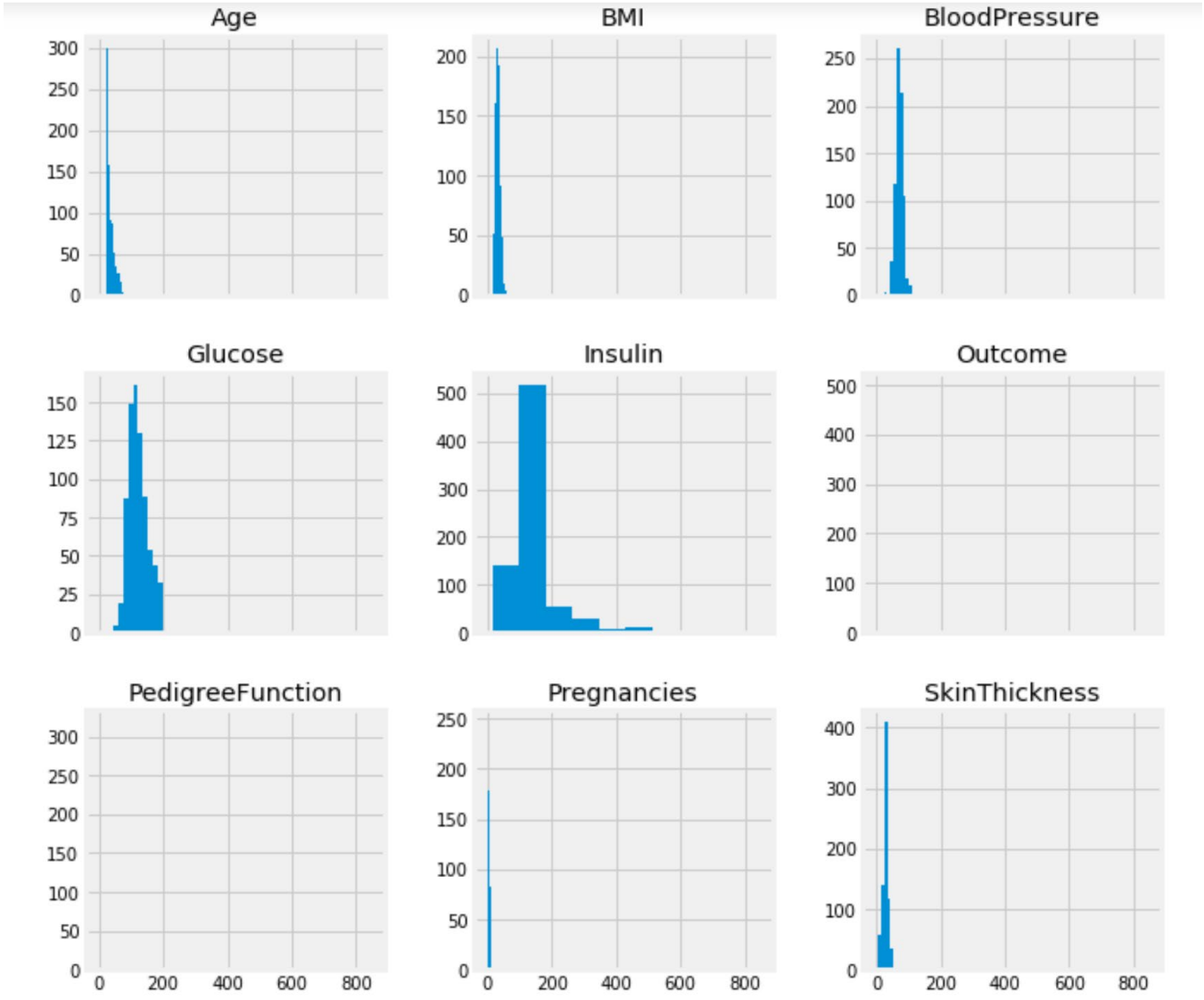
```
data_imputed_minmax_scaled.describe()
```

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	PedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	0.226180	0.501205	0.493930	0.240798	0.170130	0.291564	0.168179	0.204015	0.348958
std	0.198210	0.196361	0.123432	0.095554	0.102189	0.140596	0.141473	0.196004	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	0.058824	0.359677	0.408163	0.195652	0.129207	0.190184	0.070773	0.050000	0.000000
50%	0.176471	0.470968	0.491863	0.240798	0.170130	0.290389	0.125747	0.133333	0.000000
75%	0.352941	0.620968	0.571429	0.271739	0.170130	0.376278	0.234095	0.333333	1.000000
max	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000	1.000000

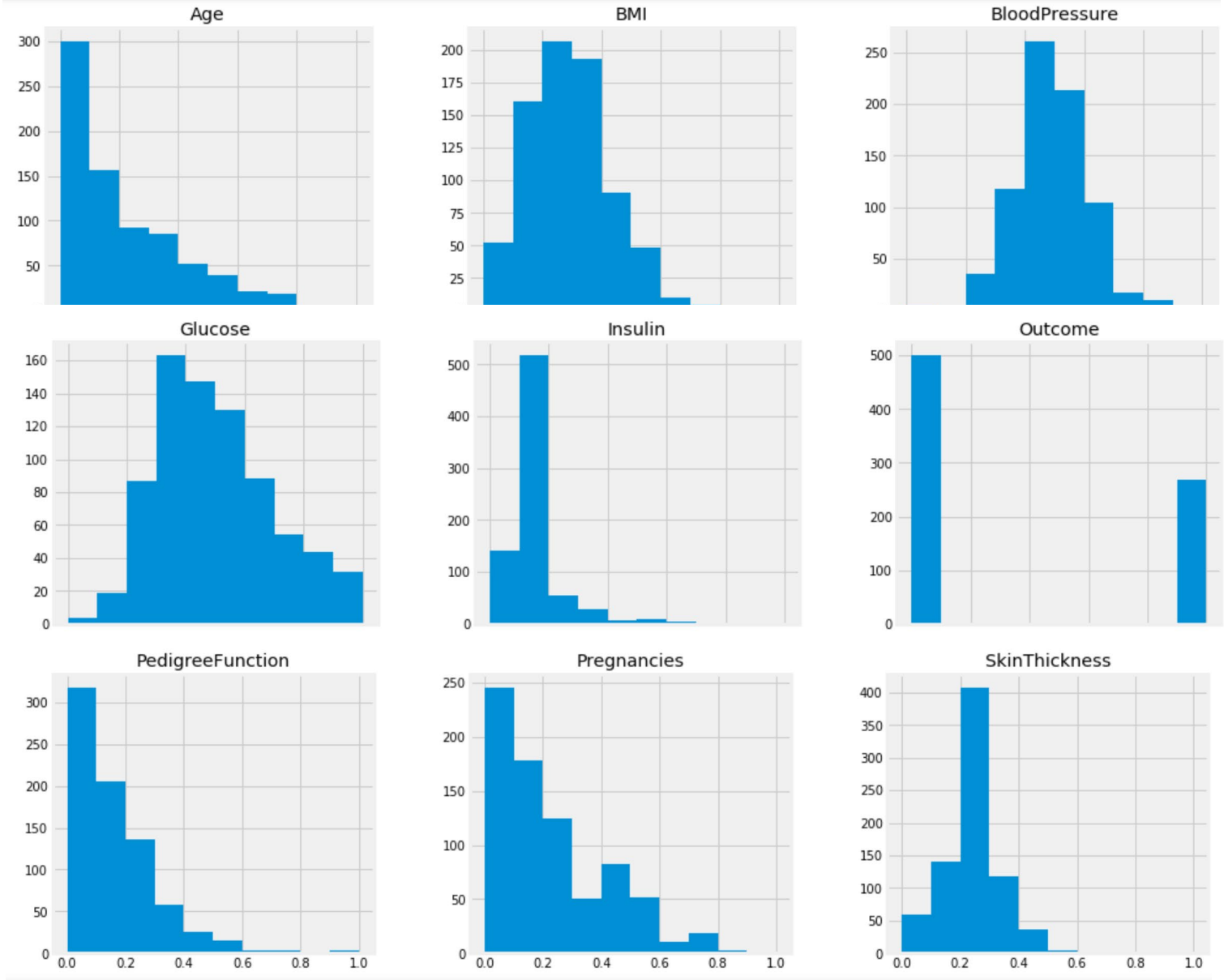
```
data_imputed_mean.describe()
```

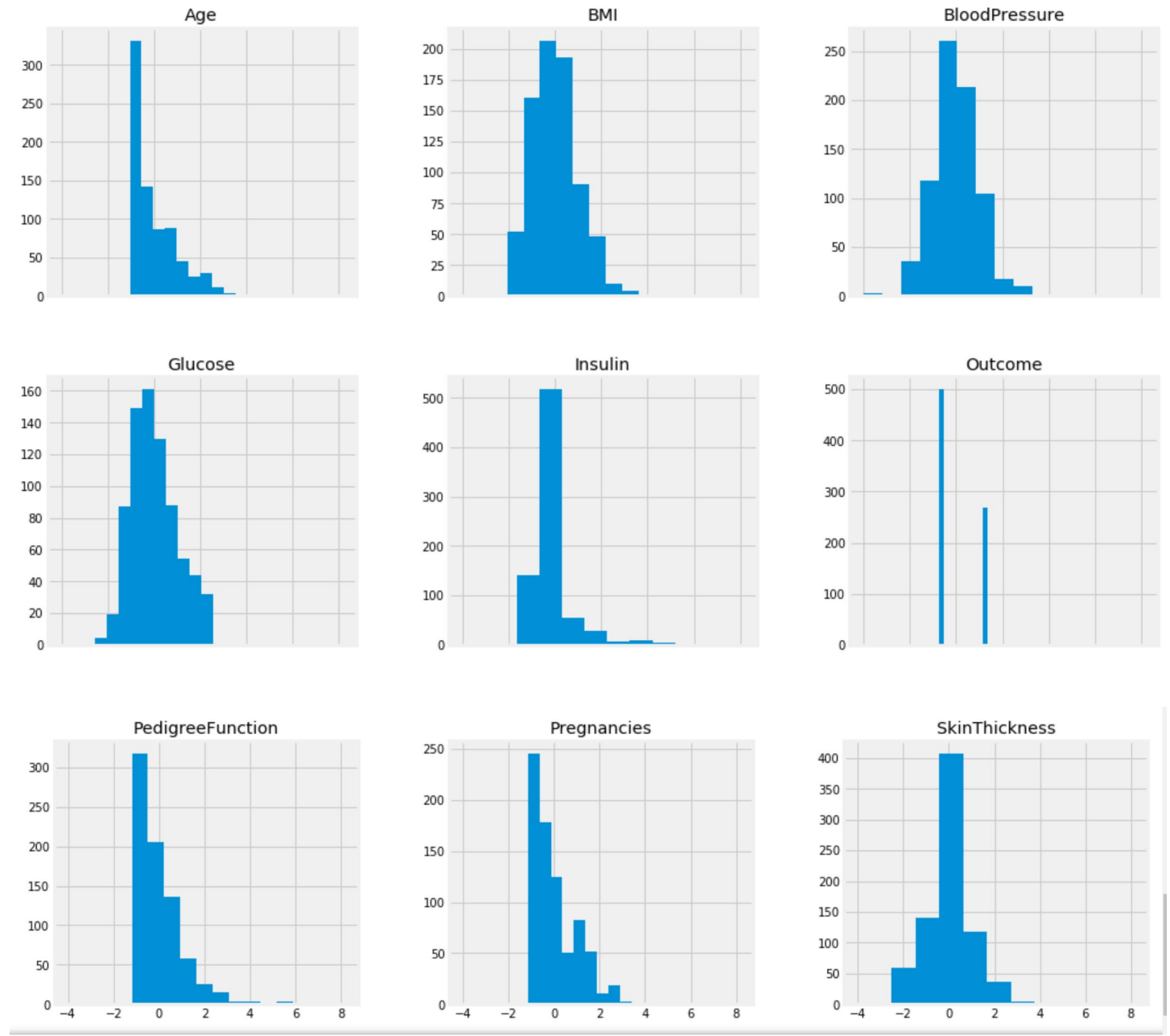
	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	PedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	121.686763	72.405184	29.153420	155.548223	32.457464	0.471876	33.240885	0.348958
std	3.369578	30.435949	12.096346	8.790942	85.021108	6.875151	0.331329	11.760232	0.476951
min	0.000000	44.000000	24.000000	7.000000	14.000000	18.200000	0.078000	21.000000	0.000000
25%	1.000000	99.750000	64.000000	25.000000	121.500000	27.500000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.202592	29.153420	155.548223	32.400000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	155.548223	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000


Age
BMI
BloodPressure
Glucose
Insulin
Outcome
PedigreeFunction
Pregnancies
SkinThickness



It is clear that each column uses a different scale weight.
Normalization ensures that all values are treated equally for the model.


Age
BMI
BloodPressure
Glucose
Insulin
Outcome
PedigreeFunction
Pregnancies
SkinThickness


Age
BMI
BloodPressure
Glucose
Insulin
Outcome
PedigreeFunction
Pregnancies
SkinThickness

Model Accuracy Before and After Dealing with Missing values and Normalizing the Dataset

	# of rows a model learned from	Cross-validation accuracy
Drop missing-valued rows	391	.7449
Impute values with 0	768	.7304
Impute values with mean of column	768	.7318
Impute values with median of column	768	.7357
Z-score normalization with mean imputing	768	.7422
Min-max normalization with mean imputing	**768**	**.7461**