# Module 4: Data Preprocessing

- Reference the "Data" chapter of the "Introduction to Data Mining" book to understand some of the concepts introduced in this tutorial.
- Down notebook

## 4.1 Data Quality Issues

- Poor data quality can have an adverse effect on data mining.
- Classes of quality issue: noise,outliers,missing values, and duplicate data.
- Target of this session : alleviate some of these data quality problems.

```
# download the dataset using Pandas read_csv() function
# and display its first 5 data points
import pandas as pd

data = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/breast-cancer-wisconsin.data', header=None)
data.columns = ['Sample code', 'Clump Thickness', 'Uniformity of Cell Size', 'Uniformity of CellShape','Marginal Adhesion', 'Single Epithelial Cell Size', 'Bare Nuclei', 'Bland Chromatin','Normal Nucleoli', 'Mitoses','Class']
data = data.drop(['Sample code'],axis=1)
print('Number of instances = %d' % (data.shape[0])) # display the number of rows
print('Number of attributes = %d' % (data.shape[1])) # display the number of columns
data.head()
```

```
Number of instances = 699
Number of attributes = 10
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

| | Clump Thickness | Uniformity of Cell Size | Uniformity of CellShape | Marginal Adhesion | Single Epithelial Cell Size | Bare Nuclei | Bland Chromatin | Normal Nucleoli | Mitoses | Class |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 5 | 1 | 1 | 1 | 2 | 1 | 3 | 1 | 1 | 2 |
| **1** | 5 | 4 | 4 | 5 | 7 | 10 | 3 | 2 | 1 | 2 |
| **2** | 3 | 1 | 1 | 1 | 2 | 2 | 3 | 1 | 1 | 2 |
| **3** | 6 | 8 | 8 | 1 | 3 | 4 | 3 | 7 | 1 | 2 |
| **4** | 4 | 1 | 1 | 3 | 2 | 1 | 3 | 1 | 1 | 2 |

### (1) Missing Values

- It is not unusual for an object to be missing one or more attribute values :
  1. the information was not collected;

2. some attributes are inapplicable to the data instances.

- The missing values are encoded as '?' in the original data.

## Instead of replacing the missing values

### 1st. approach : convert ? to NaN

```
1. Convert the missing value to NaN;
2. Count the number of missing values in each column of the data.
```

### 2nd. approach : discard the data points that contain missing values

- function : dropna()

```
# 1st. approach
import numpy as np

data = data.replace('?', np.NaN)

print('Number of instances = %d' % (data.shape[0]))
print('Number of attributes = %d' % (data.shape[1]))

print('Number of missing values:')
for col in data.columns:
    print('\t%s: %d'%(col,data[col].isna().sum()))

'''
The result will show that only the 'Bare Nuclei' column contains missing values.
'''
```

```
Number of instances = 699
Number of attributes = 10
Number of missing values:
    Clump Thickness: 0
    Uniformity of Cell Size: 0
    Uniformity of CellShape: 0
    Marginal Adhesion: 0
    Single Epithelial Cell Size: 0
    Bare Nuclei: 16
    Bland Chromatin: 0
    Normal Nucleoli: 0
    Mitoses: 0
    Class: 0
```

```
"\nThe result will show that only the 'Bare Nuclei' column contains missing values.\n"
```

```
data2 = data['Bare Nuclei']

print('Before replacing missing values:')
print(data2[20:25])

print('\nAfter replacing missing values:')
data2 = data2.fillna(data2.median())
print(data2[20:25])

'''
The missing values in the 'Bare Nuclei' column are replaced by the median value of that column.
The values before and after replacement are shown for a subset of the data points.
'''
```

```
Before replacing missing values:
20     10
21      7
22      1
23    NaN
```

```
24      1
Name: Bare Nuclei, dtype: object

After replacing missing values:
20     10
21      7
22      1
23    1.0
24      1
Name: Bare Nuclei, dtype: object
```

```
"\nThe missing values in the 'Bare Nuclei' column are replaced by the median value of that column. \nThe values before and after replacement are shown for a subset of the data points.\n"
```

```
print("Number of rows in original data = %d" % (data.shape[0]))

data2 = data.dropna()
print("Number of rows after discarding missing values = %d" % (data2.shape[0]))
```

```
Number of rows in original data = 699
Number of rows after discarding missing values = 683
```

```
# 2nd. approach

print('Number of rows in original data = %d' %(data.shape[0]))

data2.dropna()
print("Number of rows after discarding missing values = %d" % (data2.shape[0]))
```

```
Number of rows in original data = 699
Number of rows after discarding missing values = 683
```

## (2) Outliers

- Outliers are data instances with characteristics that are considerably different from the rest of the dataset.

### Identify the columns contain outliers

- In the example code below, we will draw a boxplot to identify the columns in the table that contain outliers.
- Note that the values in all columns (except for 'Bare Nuclei') are originally stored as 'int64' whereas the values in the 'Bare Nuclei' column are stored as string objects (since the column initially contains strings such as '?' for representing missing values).
- Thus, we must convert the column into numeric values first before creating the boxplot.
- Otherwise, the column will not be displayed when drawing the boxplot.

### Discard the outliers

- 1st. Step : Compute the Z-score for each attribute;
- 2nd. Step : Remove those instances containing attributes with abnormally high or low Z - score(assume value of n, Z > n or Z <= -n).

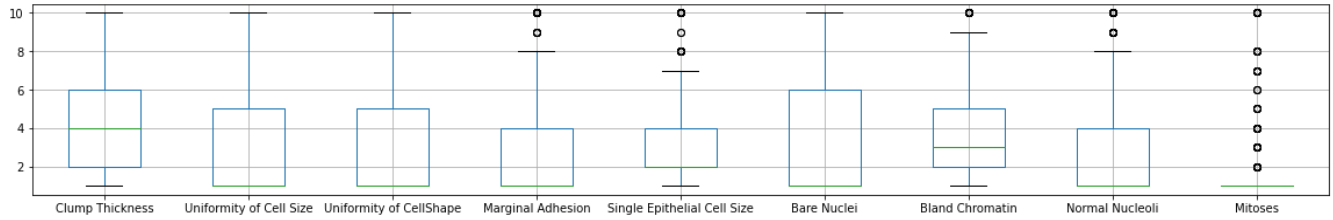
### Function

- boxplot():
    - figsize = ( 1st., 2nd.) : the size of graph, 1st. - height, 2nd. - width

```
%matplotlib inline

data2 = data.drop(['Class'], axis = 1)
data2['Bare Nuclei'] = pd.to_numeric(data2['Bare Nuclei'])
data2.boxplot(figsize=(20,3))
```

```
<AxesSubplot:>
```




The boxplots suggest that only 5 of the columns (Marginal Adhesion, Single Epithetial Cell Size, Bland Cromatin, Normal Nucleoli, and Mitoses) contain abnormally high values.

To discard the outliers, we can compute the Z-score for each attribute and remove those instances containing attributes with abnormally high or low Z-score (e.g., if Z > 3 or Z <= -3).

## (3) Standardization

- Note : The missing values (NaN) are not affected by the standardization process.

### Function

- std(): get the standardization of

```
# The following code shows the results of standardizing the columns of the data.

Z = (data2 - data2.mean())/data2.std()
Z[20:25]
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

| | Clump Thickness | Uniformity of Cell Size | Uniformity of CellShape | Marginal Adhesion | Single Epithelial Cell Size | Bare Nuclei | Bland Chromatin | Normal Nucleoli | Mitoses |
|---|---|---|---|---|---|---|---|---|---|
| **20** | 0.917080 | -0.044070 | -0.406284 | 2.519152 | 0.805662 | 1.771569 | 0.640688 | 0.371049 | 1.405526 |
| **21** | 1.982519 | 0.611354 | 0.603167 | 0.067638 | 1.257272 | 0.948266 | 1.460910 | 2.335921 | -0.343666 |
| **22** | -0.503505 | -0.699494 | -0.742767 | -0.632794 | -0.549168 | -0.698341 | -0.589645 | -0.611387 | -0.343666 |
| **23** | 1.272227 | 0.283642 | 0.603167 | -0.632794 | -0.549168 | NaN | 1.460910 | 0.043570 | -0.343666 |
| **24** | -1.213798 | -0.699494 | -0.742767 | -0.632794 | -0.549168 | -0.698341 | -0.179534 | -0.611387 | -0.343666 |

```
# The results of discarding columns with Z > 3 or Z <= -3.

print("Number of rows before discarding outliers = %d" % (Z.shape[0]))

Z2 = Z.loc[((Z > -3).sum(axis=1)==9) & ((Z <= 3).sum(axis=1)==9),:]
print("Number of rows after discarding missing values = %d" % (Z2.shape[0]))
```

```
Number of rows before discarding outliers = 699
Number of rows after discarding missing values = 632
```

# (4) Duplicate Data

- Some datasets, especially those obtained by merging multiple data sources, may contain duplicates or near duplicate instances.
- The term deduplication is often used to refer to the process of dealing with duplicate data issues.

## Function

- duplicated() : return a Boolean array that indicates whether each row is a duplicate of a previous row in the table.

## Check Duplicate Instance

```
# first check for duplicate instances in the breast cancer dataset.
dups = data.duplicated()
dups
```

```
0      False
1      False
2      False
3      False
4      False
       ...
694    False
695     True
696    False
697    False
698    False
Length: 699, dtype: bool
```

```
print("Number of duplicate rows = %d" % (dups.sum()))
data.loc[[11,28]]
```

```
Number of duplicate rows = 236
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

| | Clump Thickness | Uniformity of Cell Size | Uniformity of CellShape | Marginal Adhesion | Single Epithelial Cell Size | Bare Nuclei | Bland Chromatin | Normal Nucleoli | Mitoses | Class |
|---|---|---|---|---|---|---|---|---|---|---|
| **11** | 2 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 2 |
| **28** | 2 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 2 |

The results suggest there are 236 duplicate rows in the breast cancer dataset.

For example, the instance with row index 11 has identical attribute values as the instance with row index 28.

Although such duplicate rows may correspond to samples for different individuals, in this hypothetical example, we assume that the duplicates are samples taken from the same individual and illustrate below how to remove the duplicated rows.

### Remove Duplicated Instances

### Function

- drop_duplicates() : remove duplicated instances

```
print("Number of rows before discarding duplicates = %d" % (data.shape[0]))

data2 = data.drop_duplicates()
print("Number of rows after discarding duplicates = %d" % (data2.shape[0]))
```

```
Number of rows before discarding duplicates = 699
Number of rows after discarding duplicates = 463
```

## 4.2 Aggregation

- Data aggregation is a preprocessing task where the values of two or more objects are combined into a single object.
- Motivation :
  1. reducing the size of data to be processed;
  2. changingthe granularity of analysis (from fine-scale to coarser-scale);
  3. improving the stability of the data.
- Example:

  In the example below, we will use the daily precipitation time series data for a weather station located at Detroit Metro Airport.

  The raw data was obtained from the Climate Data Online website.

  The daily precipitation time series will be compared against its monthly values.

### Function

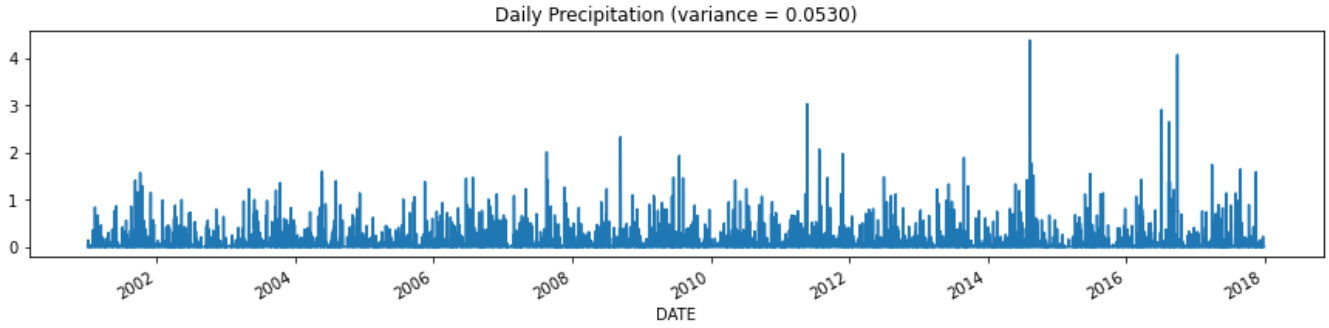
- groupby() : combine dispersive objects into a sigle object
  - pd.Grouper(freq = 'M') : set the frequency as Month

```
# Load the precipitation time series data
daily = pd.read_csv('DTW_prec.csv',header='infer')

# Draw a line plot of its daily time series.
daily.index = pd.to_datetime(daily['DATE'])
daily = daily['PRCP']

ax = daily.plot(kind = 'line',figsize=(15,3))
ax.set_title('Daily Precipitation (variance = %.4f)' % (daily.var()))
```

```
Text(0.5, 1.0, 'Daily Precipitation (variance = 0.0530)')
```




Observe that the daily time series appear to be quite chaotic and varies significantly from one time step to another.

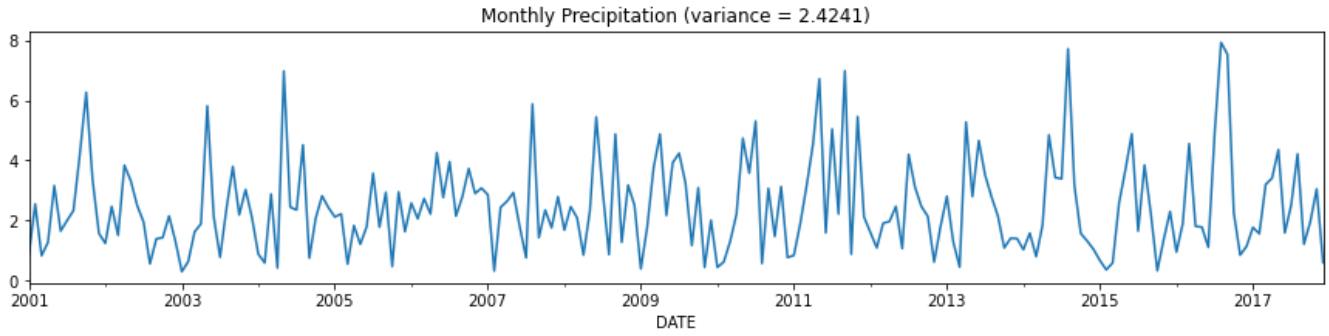
The time series can be grouped and aggregated by month to obtain the total monthly precipitationvalues.

The resulting time series appears to vary more smoothly compared to the daily time series.

```
# Draw a line plot of its monthly time series.
monthly = daily.groupby(pd.Grouper(freq='M')).sum()

ax = monthly.plot(kind = 'line', figsize=(15,3))
ax.set_title('Monthly Precipitation (variance = %.4f)' % (monthly.var()))
```

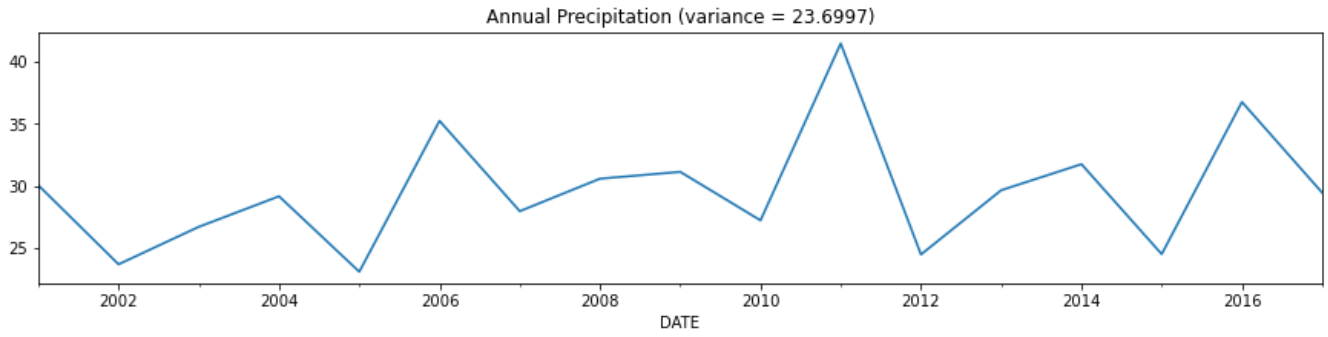
```
Text(0.5, 1.0, 'Monthly Precipitation (variance = 2.4241)')
```




The daily precipitation time series are grouped and aggregated by year to obtain the annual precipitation values.

```
# Draw a line plot of its yearly time series
annual = daily.groupby(pd.Grouper(freq='Y')).sum()
ax = annual.plot(kind='line',figsize=(15,3))
ax.set_title('Annual Precipitation (variance = %.4f)' % (annual.var()))
```

```
Text(0.5, 1.0, 'Annual Precipitation (variance = 23.6997)')
```




## 4.3 Sampling

- An approach commonly used to facilitate :
    1. data reduction for exploratory data analysis and scaling up algorithms to big data applications;
    2. quantifying uncertainties due to varying data distributions.
- Methods:
    1. sampling without replacement : each selected instance is removed from the dataset;
    2. sampling with replacement : each selected instance is not removed.

## Function

- sample() : select the samples
  - n : set the number of samples
  - frac :
  - random_state : specifies the seed value of the random number generator.

In the example below, we will apply sampling with replacement and without replacement to the breast cancer dataset obtained from the UCI machine learning repository.

```
# Initially display the first five records of the table
data.head()
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

| | Clump Thickness | Uniformity of Cell Size | Uniformity of CellShape | Marginal Adhesion | Single Epithelial Cell Size | Bare Nuclei | Bland Chromatin | Normal Nucleoli | Mitoses | Class |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 5 | 1 | 1 | 1 | 2 | 1 | 3 | 1 | 1 | 2 |
| **1** | 5 | 4 | 4 | 5 | 7 | 10 | 3 | 2 | 1 | 2 |
| **2** | 3 | 1 | 1 | 1 | 2 | 2 | 3 | 1 | 1 | 2 |
| **3** | 6 | 8 | 8 | 1 | 3 | 4 | 3 | 7 | 1 | 2 |
| **4** | 4 | 1 | 1 | 3 | 2 | 1 | 3 | 1 | 1 | 2 |

```
# a sample of size 3 is randomly selected
# (without replacement) from the original data.
sample = data.sample(n=3)
sample
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

| | Clump Thickness | Uniformity of Cell Size | Uniformity of CellShape | Marginal Adhesion | Single Epithelial Cell Size | Bare Nuclei | Bland Chromatin | Normal Nucleoli | Mitoses | Class |
|---|---|---|---|---|---|---|---|---|---|---|
| **620** | 3 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 2 |
| **113** | 10 | 10 | 10 | 3 | 10 | 8 | 8 | 1 | 1 | 4 |
| **443** | 1 | 1 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 2 |

```
# randomly select 1% of the data (without replacement)
# and display the selected samples.
sample = data.sample(frac=0.01, random_state=1)
sample
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

| | Clump Thickness | Uniformity of Cell Size | Uniformity of CellShape | Marginal Adhesion | Single Epithelial Cell Size | Bare Nuclei | Bland Chromatin | Normal Nucleoli | Mitoses | Class |
|---|---|---|---|---|---|---|---|---|---|---|
| **584** | 5 | 1 | 1 | 6 | 3 | 1 | 1 | 1 | 1 | 2 |
| **417** | 1 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 2 |
| **606** | 4 | 1 | 1 | 2 | 2 | 1 | 1 | 1 | 1 | 2 |
| **349** | 4 | 2 | 3 | 5 | 3 | 8 | 7 | 6 | 1 | 4 |
| **134** | 3 | 1 | 1 | 1 | 3 | 1 | 2 | 1 | 1 | 2 |
| **502** | 4 | 1 | 1 | 2 | 2 | 1 | 2 | 1 | 1 | 2 |
| **117** | 4 | 5 | 5 | 10 | 4 | 10 | 7 | 5 | 8 | 4 |

Finally, we perform a sampling with replacement to create a sample whose size is equal to 1% of the entiredata.

```
# observe duplicate instances in the sample
# by increasing the sample size.
sample = data.sample(frac=0.01, replace=True,random_state=1)
sample
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

| | Clump Thickness | Uniformity of Cell Size | Uniformity of CellShape | Marginal Adhesion | Single Epithelial Cell Size | Bare Nuclei | Bland Chromatin | Normal Nucleoli | Mitoses | Class |
|---|---|---|---|---|---|---|---|---|---|---|
| **37** | 6 | 2 | 1 | 1 | 1 | 1 | 7 | 1 | 1 | 2 |
| **235** | 3 | 1 | 4 | 1 | 2 | NaN | 3 | 1 | 1 | 2 |
| **72** | 1 | 3 | 3 | 2 | 2 | 1 | 7 | 2 | 1 | 2 |
| **645** | 3 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 2 |
| **144** | 2 | 1 | 1 | 1 | 2 | 1 | 2 | 1 | 1 | 2 |
| **129** | 1 | 1 | 1 | 1 | 10 | 1 | 1 | 1 | 1 | 2 |
| **583** | 3 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 2 |

## 4.4 Discretization

- Discretization is a data preprocessing step that is often used to transform a continuous-valued attribute to a categorical attribute.

## Function

- value_counts() : applied to count the frequency of each attribute value.
- cut() : discretize the attribute into 4 bins of similar interval widths.
- qcut() : partition the values into 4 bins such that each bin has nearly the same number of instances.

## For the equal width method

- apply the cut() function to discretize the attribute into 4 bins of similar interval widths.
- use the value_counts()function to determine the number of instances in each bin.

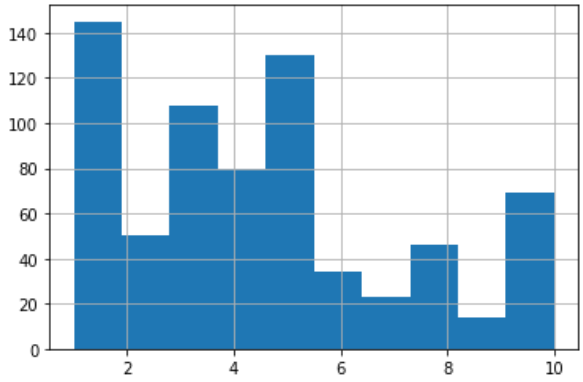
## For the equal frequency method

- apply the qcut() function to partition the values into 4 bins such that each bin has nearly the same number of instances.
- use the value_counts() function to determine the number of instances in each bin.

The example below illustrates two simple but widely-used unsupervised discretization
methods (equal width and equal depth) applied to the 'Clump Thickness' attribute of the breast cancer dataset.

```
# Plot a histogram
# shows the distribution of the attribute values
data['Clump Thickness'].hist(bins=10)   # histogram graph
data['Clump Thickness'].value_counts(sort=False) # values counts
```

```
1     145
2      50
3     108
4      80
5     130
6      34
7      23
8      46
9      14
10     69
Name: Clump Thickness, dtype: int64
```




```
# the equal width method

bins = pd.cut(data['Clump Thickness'],4)
bins.value_counts(sort=False)
```

```
(0.991, 3.25]    303
(3.25, 5.5]      210
(5.5, 7.75]       57
(7.75, 10.0]     129
Name: Clump Thickness, dtype: int64
```

```
# the equal frequency method

bins = pd.qcut(data['Clump Thickness'],4)
bins.value_counts(sort=False)
```

```
(0.999, 2.0]    195
(2.0, 4.0]      188
(4.0, 6.0]      164
(6.0, 10.0]     152
Name: Clump Thickness, dtype: int64
```

## 4.5 Principal Component Analysis (PCA)

- A classical method for reducing the number of attributes in the data by projecting the data from its original high-dimensional space into a lower-dimensional space.
- The new attributes created by PCA is also called as components.
- The properties of components:
    1. They are linear combinations of the original attributes;
    2. They are orthogonal (perpendicular) to each other;
    3. They capture the maximum amount of variation in the data.

```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

numImages = 16
fig = plt.figure(figsize=(7,7))
imgData = np.zeros(shape=(numImages,36963))

for i in range(1,numImages+1):
     filename = 'pics/Picture'+str(i)+'.jpg'
     img = mpimg.imread(filename)
     ax = fig.add_subplot(4,4,i)
     plt.imshow(img)
     plt.axis('off')
     ax.set_title(str(i))
     imgData[i-1] = np.array(img.flatten()).reshape(1,img.shape[0]*img.shape[1]*img.shape[2])

'''
The example below illustrates the application of PCA to an image dataset. There are 16 RGB files, each of which has a size of 111 x 111 pixels.

The example code below will read each image file and convert the RGB image into a 111 x 111 x 3 = 36963 feature values. This will create a data matrix of size 16 x 36963.
'''
```

```
'\nThe example below illustrates the application of PCA to an image dataset. There are 16 RGB files, each of which has a size of 111 x 111 pixels. \n\nThe example code below will read each image file and convert the RGB image into a 111 x 111 x 3 = 36963 feature values. This will create a data matrix of size 16 x 36963.\n'
```


Using PCA, the data matrix is projected to its first two principal components.

The projected values of the original image data are stored in a pandas DataFrame object named projected.

```
import pandas as pd
from sklearn.decomposition import PCA

numComponents = 2
pca = PCA(n_components=numComponents)
pca.fit(imgData)

projected = pca.transform(imgData)
projected = pd.DataFrame(projected,columns=['pc1','pc2'],index=range(1,numImages+1))
projected['food'] = ['burger', 'burger','burger','burger','drink','drink','drink','drink','pasta', 'pasta', 'pasta',
'pasta', 'chicken', 'chicken', 'chicken', 'chicken']
projected


```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

| | pc1 | pc2 | food |
|---|---|---|---|
| **1** | -1576.595378 | 6642.807108 | burger |
| **2** | -493.845769 | 6397.299481 | burger |
| **3** | 990.134297 | 7236.504610 | burger |
| **4** | 2189.844227 | 9050.455025 | burger |
| **5** | -7843.182186 | -1063.588929 | drink |
| **6** | -8498.413936 | -5438.025404 | drink |
| **7** | -11181.749777 | -5318.942146 | drink |
| **8** | -6851.939375 | 1124.318518 | drink |
| **9** | 7635.179712 | -5042.928608 | pasta |
| **10** | -708.065183 | -528.761416 | pasta |
| **11** | 7236.300598 | -5300.586052 | pasta |
| **12** | 4417.412850 | -4658.147146 | pasta |
| **13** | 11864.496600 | 1472.262617 | chicken |
| **14** | 76.453155 | 1365.873898 | chicken |
| **15** | -7505.697233 | -1164.669242 | chicken |
| **16** | 10249.667400 | -4773.872314 | chicken |

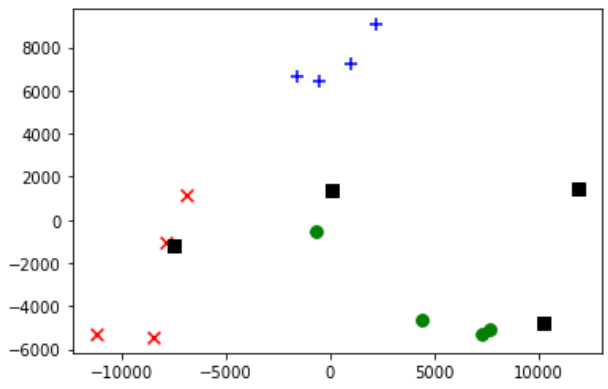
```
# draw a scatter plot to display the projected values.

import matplotlib.pyplot as plt

colors = {'burger':'b', 'drink':'r', 'pasta':'g', 'chicken':'k'}
markerTypes = {'burger':'+', 'drink':'x', 'pasta':'o', 'chicken':'s'}

for foodType in markerTypes:
    d = projected[projected['food']==foodType]
    plt.scatter(d['pc1'],d['pc2'],c=colors[foodType],s=60,marker=markerTypes[foodType])

```




Observe that the images of burgers, drinks,and pastas are all projected to the same region.

However, the images for fried chicken (shown as black squares in the diagram) are harder to discriminate.