Module 4: Data Preprocessing

Assignment 1.2

Python code 3

University of San Diego

ADS 502

Dingyi Duan

4.1 Data Quality Issues

Poor data quality can have an adverse effect on data mining. Among the common data quality issues include noise, outliers, missing values, and duplicate data. This section presents examples of Python code to alleviate some of these data quality problems. We begin with an example dataset from the UCI machine learning repository containing information about breast cancer patients. We will first download the dataset using Pandas read_csv() function and display its first 5 data points.

Code:

```
In [1]: # Import the csv file
```

In [2]:
```
import pandas as pd
data = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/bre
data.columns = ['Sample code', 'Clump Thickness', 'Uniformity of Cell Size', 'Uni
                'Marginal Adhesion', 'Single Epithelial Cell Size', 'Bare Nuclei'
                'Normal Nucleoli', 'Mitoses','Class']

data = data.drop(['Sample code'],axis=1)
print('Number of instances = %d' % (data.shape[0]))
print('Number of attributes = %d' % (data.shape[1]))
data.head()
```

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

Out[2]:

	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses	C
0	5	1	1	1	2	1	3	1	1	
1	5	4	4	5	7	10	3	2	1	
2	3	1	1	1	2	2	3	1	1	
3	6	8	8	1	3	4	3	7	1	
4	4	1	1	3	2	1	3	1	1	

4.1.1 Missing Values

It is not unusual for an object to be missing one or more attribute values. In some cases, the information was not collected; while in other cases, some attributes are inapplicable to the data instances. This section presents examples on the different approaches for handling missing values.

According to the description of the data (https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original) (https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)), the missing values are encoded as '?' in the original data. Our first task is to convert the missing values to NaNs. We can then count the number of missing values in each column of the data.

Code:

In [3]:
```
# Show missing values in each column
```

In [4]:
```
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()))
```

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

Observe that only the 'Bare Nuclei' column contains missing values. In the following example, 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.

Code:

In [5]:
```
# Fill missing values with median
```

In [6]:
```
data2 = data['Bare Nuclei']

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

print('\nAfter replacing missing values:')
print(data2[20:25])
```

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

Instead of replacing the missing values, another common approach is to discard the data points that contain missing values. This can be easily accomplished by applying the dropna() function to the data frame.

Code:

In [7]:
```
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
```

4.1.2 Outliers

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

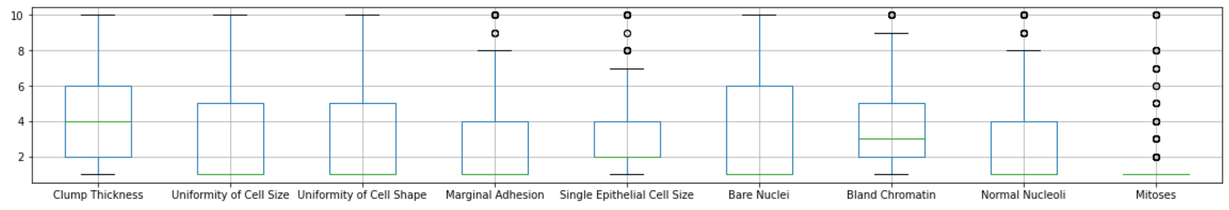
Code:

In [8]:
```
# Use boxplot to show outliers
```

In [9]:
```
%matplotlib inline

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

Out[9]:
```
<AxesSubplot:>
```



The boxplots suggest that only 5 of the columns (Marginal Adhesion, Single Epithelial 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).

Code:

The following code shows the results of standardizing the columns of the data. Note that missing values (NaN) are not affected by the standardization process.

In [10]:
```
# Normalization/Standardization
```

In [11]:
```
Z = (data2-data2.mean())/data2.std()
Z[20:25]
```

Out[11]:

	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mito
20	0.917080	-0.044070	-0.406284	2.519152	0.805662	1.771569	0.640688	0.371049	1.405
21	1.982519	0.611354	0.603167	0.067638	1.257272	0.948266	1.460910	2.335921	-0.343
22	-0.503505	-0.699494	-0.742767	-0.632794	-0.549168	-0.698341	-0.589645	-0.611387	-0.343
23	1.272227	0.283642	0.603167	-0.632794	-0.549168	NaN	1.460910	0.043570	-0.343
24	-1.213798	-0.699494	-0.742767	-0.632794	-0.549168	-0.698341	-0.179534	-0.611387	-0.343

Code:

The following code shows the results of discarding columns with Z > 3 or Z <= -3.

In [12]:
```
# Outliers = Z value > 3 or < -3
```

In [13]:
```
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
```

4.1.3 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.

Code:

In the following example, we first check for duplicate instances in the breast cancer dataset.

In [14]:
```
# Use duplicated to show duplicates
```

In [15]:
```
dups = data.duplicated()
print('Number of duplicate rows = %d' % (dups.sum()))
data.loc[[11,28]]
```

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

Out[15]:

	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses
11	2	1	1	1	2	1	2	1	1
28	2	1	1	1	2	1	2	1	1

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

Code:

In [16]:
```
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. The motivation for aggregation includes (1) reducing the size of data to be processed, (2) changing the granularity of analysis (from fine-scale to coarser-scale), and (3) improving the stability of the data.

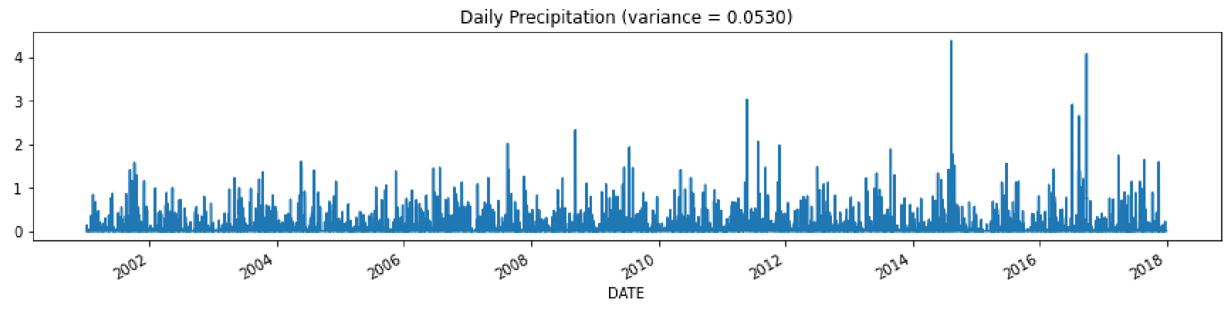
In the example below, we will use the daily precipitation time series data for a weather station located at Detroit Metro Airport. The raw data was obtained from the Climate Data Online website (https://www.ncdc.noaa.gov/cdo-web/ (https://www.ncdc.noaa.gov/cdo-web/)). The daily precipitation time series will be compared against its monthly values.

Code:

The code below will load the precipitation time series data and draw a line plot of its daily time series.

In [17]:
```
daily = pd.read_csv('DTW_prec.csv', header='infer')
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()))
```

Out[17]:
```
Text(0.5, 1.0, 'Daily Precipitation (variance = 0.0530)')
```

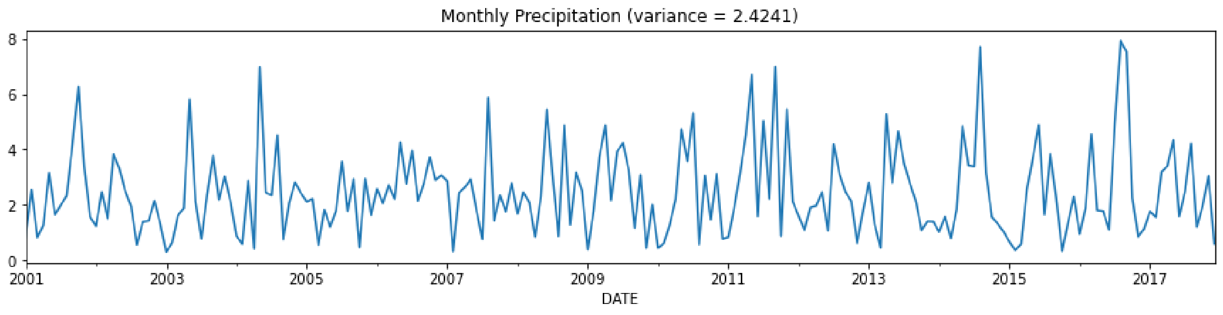


Observe that the daily time series appear to be quite chaotic and varies significantly from one time step to another. The time series can be grouped and aggregated by month to obtain the total monthly precipitation values. The resulting time series appears to vary more smoothly compared to the daily time series.

Code:

```
In [18]: 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()))
```

```
Out[18]: Text(0.5, 1.0, 'Monthly Precipitation (variance = 2.4241)')
```

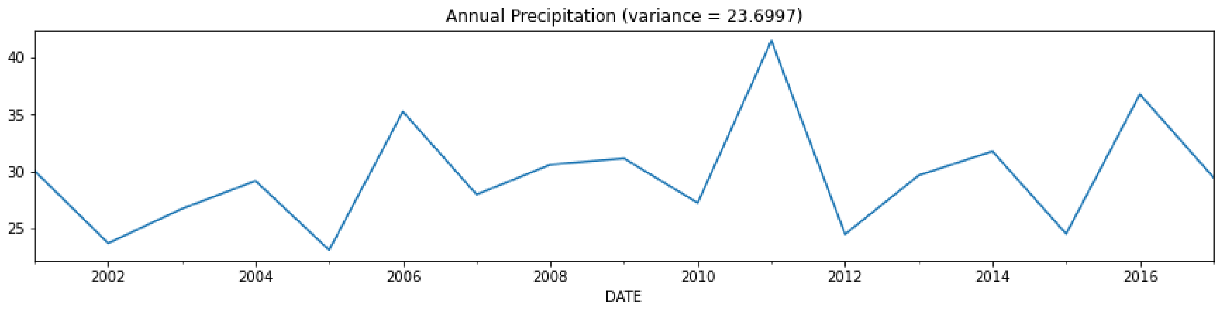


In the example below, the daily precipitation time series are grouped and aggregated by year to obtain the annual precipitation values.

Code:

```
In [19]: 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()))
```

```
Out[19]: Text(0.5, 1.0, 'Annual Precipitation (variance = 23.6997)')
```



4.3 Sampling

Sampling is an approach commonly used to facilitate (1) data reduction for exploratory data analysis and scaling up algorithms to big data applications and (2) quantifying uncertainties due to varying data distributions. There are various methods available for data sampling, such as sampling without replacement, where each selected instance is removed from the dataset, and sampling with replacement, where each selected instance is not removed, thus allowing it to be selected more than once in the sample.

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.

Code:

We initially display the first five records of the table.

In [20]:
```
data.head()
```

Out[20]:

	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses	C
0	5	1	1	1	2	1	3	1	1	
1	5	4	4	5	7	10	3	2	1	
2	3	1	1	1	2	2	3	1	1	
3	6	8	8	1	3	4	3	7	1	
4	4	1	1	3	2	1	3	1	1	

In the following code, a sample of size 3 is randomly selected (without replacement) from the original data.

Code:

In [21]:
```
# Randomly select 3 rows using "sample"
```

In [22]:
```
sample = data.sample(n=3)
sample
```

Out[22]:

	Clump Thickness	Uniformity of Cell Size	Uniformity of Cell Shape	Marginal Adhesion	Single Epithelial Cell Size	Bare Nuclei	Bland Chromatin	Normal Nucleoli	Mitoses
399	1	2	3	1	2	1	1	1	1
532	1	1	1	1	1	1	3	1	1
667	3	1	1	1	2	1	3	1	1

In the next example, we randomly select 1% of the data (without replacement) and display the selected samples. The random_state argument of the function specifies the seed value of the random number generator.

Code:

In [23]:
```
sample = data.sample(frac=0.01, random_state=1)
sample
```

Out[23]:

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

Finally, we perform a sampling with replacement to create a sample whose size is equal to 1% of the entire data. You should be able to observe duplicate instances in the sample by increasing the sample size.

Code:

In [24]:
```
sample = data.sample(frac=0.01, replace=True, random_state=1)
sample
```

Out[24]:

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

4.4 Discretization

Discretization is a data preprocessing step that is often used to transform a continuous-valued attribute to a categorical attribute. The example below illustrates two simple but widely-used unsupervised discretization methods (equal width and equal depth) applied to the 'Clump

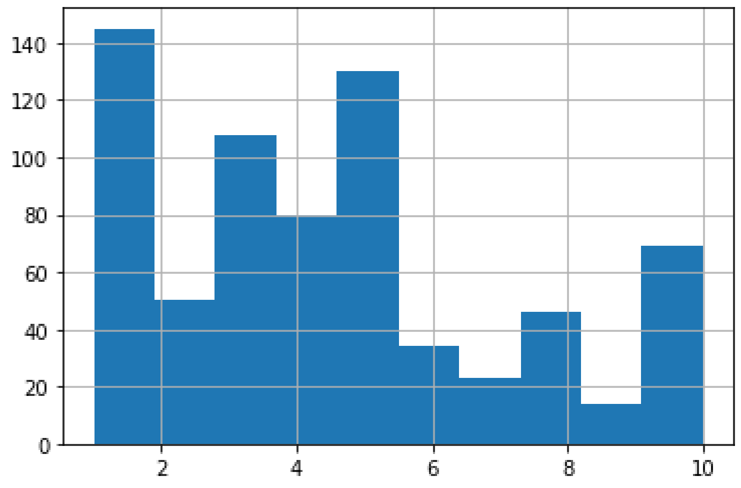
Thickness' attribute of the breast cancer dataset.

First, we plot a histogram that shows the distribution of the attribute values. The value_counts() function can also be applied to count the frequency of each attribute value.

Code:

In [25]:
```
data['Clump Thickness'].hist(bins=10)
data['Clump Thickness'].value_counts(sort=False)
```

Out[25]:
```
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
```



For the equal width method, we can apply the cut() function to discretize the attribute into 4 bins of similar interval widths. The value_counts() function can be used to determine the number of instances in each bin.

Code:

In [26]:
```
bins = pd.cut(data['Clump Thickness'],4)
bins.value_counts(sort=False)
```

Out[26]:
```
(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
```

For the equal frequency method, the qcut() function can be used to partition the values into 4 bins such that each bin has nearly the same number of instances.

Code:

```
In [27]: bins = pd.qcut(data['Clump Thickness'],4)
         bins.value_counts(sort=False)
```

```
Out[27]: (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

Principal component analysis (PCA) is 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 (also known as components) created by PCA have the following properties: (1) they are linear combinations of the original attributes, (2) they are orthogonal (perpendicular) to each other, and (3) they capture the maximum amount of variation in the data.

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.

Code:

In [28]:
```
%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 = 'C:/Users/DDY/Desktop/2021-Spring-textbooks/ADS-502/Module1/Weekly
    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]*in
```

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.

Code:

In [29]:
```
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
projected['food'] = ['burger', 'burger','burger','burger','drink','drink','drink'
                      'pasta', 'pasta', 'pasta', 'pasta', 'chicken', 'chicken', '
projected
```

Out[29]:

	pc1	pc2	food
1	-1576.715736	6640.396927	burger
2	-493.827697	6395.818552	burger
3	990.072326	7237.111036	burger
4	2189.971227	9054.644483	burger
5	-7843.037355	-1060.618926	drink
6	-8498.416971	-5437.666733	drink
7	-11181.774944	-5318.104175	drink
8	-6852.027367	1119.958114	drink
9	7635.099638	-5045.474364	pasta
10	-708.089548	-530.139540	pasta
11	7236.285430	-5300.352391	pasta
12	4417.324232	-4660.478783	pasta
13	11864.464791	1470.384582	chicken
14	76.491380	1367.044598	chicken
15	-7505.648874	-1163.492425	chicken
16	10249.829469	-4769.030955	chicken

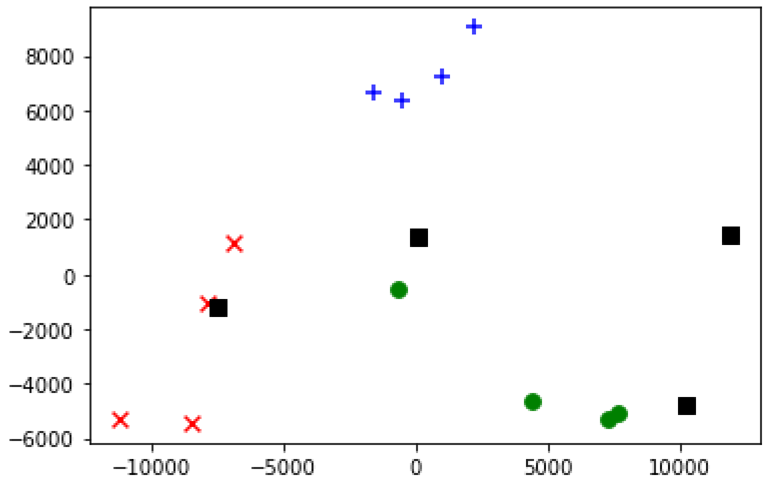
Finally, we draw a scatter plot to display the projected values. Observe that the images of burgers, drinks, and pastas are all projected to the same region. However, the images for fried chicken (shown as black squares in the diagram) are harder to discriminate.

Code:

In [30]:
```
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[fooc
```



4.6 Summary

This tutorial presents Python programming examples for data preprocessing, including data cleaning (to handle missing values and remove outliers as well as duplicate data), aggregation, sampling, discretization, and dimensionality reduction using principal component analysis.

References:

1. Lichman, M. (2013). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml] (http://archive.ics.uci.edu/ml%5D). Irvine, CA: University of California, School of Information and Computer Science.
2. Mangasarian, O.L. and Wolberg, W. H. (1990). "Cancer diagnosis via linear programming", SIAM News, Volume 23, Number 5, pp 1 & 18.
3. Wolberg, W.H. and Mangasarian, O.L. (1990). "Multisurface method of pattern separation for medical diagnosis applied to breast cytology", Proceedings of the National Academy of Sciences, U.S.A., Volume 87, pp 9193-9196.
4. Climate Data Online [https://www.ncdc.noaa.gov/cdo-web/] (https://www.ncdc.noaa.gov/cdo-web/%5D).