3803ICT
Data Analytics

Lab 03 – Data preprocessing

<u>Course Convenor:</u>
Dr. Henry Nguyen

Trimester 1 - 2018

Table of Contents

I. Exploring your data

1. Diagnose data for cleaning

1.1 Loading and view your data

In this lab, you're going to look at a subset of the Department of Buildings Job Application Filings dataset from the NYC Open Data portal. This dataset consists of job applications filed on January 22, 2017.

Import pandas

```
In [1]: import pandas as pd
```

Your first task is to load this dataset into a DataFrame and then inspect it using the .head() and .tail() methods. However, you'll find out that the printed results don't allow you to see everything you need, since there are too many columns. Therefore, you need to look at the data in another way.

Read the file into a DataFrame: df

```
In [2]: df = pd.read_csv('dob_job_application_filings_subset.csv',low_memory=False)
```

Print the head of df

```
In [3]: print(df.head())
```

```
         Job #  Doc #        Borough House #                      Street N
ame  \
0  121577873      2      MANHATTAN     386  PARK AVENUE SOUTH

1  520129502      1  STATEN ISLAND     107  KNOX PLACE
```

Print the tail of df

```
In [4]: print(df.tail())
```

```
             Job #  Doc #        Borough House #  \
12841  520143988      1  STATEN ISLAND       8
12842  121613833      1      MANHATTAN     724
```

Print the shape of df

```
In [5]: print(df.shape)
```

```
(12846, 82)
```

Print the columns of df

```
In [6]: print(df.columns)
```

```
Index(['Job #', 'Doc #', 'Borough', 'House #', 'Street Name', 'Block', 'L
ot',
```

1.2 Further diagnosis

The .info() method provides important information about a DataFrame, such as the number of rows, columns, number of non-missing values in each column, and the data type stored in each column.

```
In [7]: print(df.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12846 entries, 0 to 12845
Data columns (total 82 columns):
Job #                           12846 non-null int64
Doc #                           12846 non-null int64
Borough                         12846 non-null object
```

You'll now use the .describe() method to calculate summary statistics of your data.

```
In [8]: df.describe()
```

Out[8]:

	Job #	Doc #	Block	Lot	Bin #	Cluster	PC Filed	Zo
count	1.284600e+04	12846.000000	12846.000000	12846.000000	1.284600e+04	0.0	0.0	1.28
mean	2.426788e+08	1.162930	2703.834735	623.303441	2.314997e+06	NaN	NaN	1.43
std	1.312507e+08	0.514937	3143.002812	2000.934794	1.399062e+06	NaN	NaN	3.86
min	1.036438e+08	1.000000	1.000000	0.000000	1.000003e+06	NaN	NaN	0.00
25%	1.216206e+08	1.000000	836.000000	12.000000	1.035728e+06	NaN	NaN	0.00
50%	2.202645e+08	1.000000	1411.500000	32.000000	2.004234e+06	NaN	NaN	0.00
75%	3.208652e+08	1.000000	3355.000000	59.000000	3.343823e+06	NaN	NaN	0.00
max	5.400246e+08	9.000000	99999.000000	9078.000000	5.864852e+06	NaN	NaN	2.87

2. Exploratory data analysis

Frequency counts for categorical data

The .describe() can only be used on numeric columns. So how can you diagnose data issues when you have categorical data? One way is by using the .value_counts() method, which returns the frequency counts for each unique value in a column!

This method also has an optional parameter called dropna which is True by default. What this means is if you have missing data in a column, it will not give a frequency count of them. You want to set the dropna column to False so if there are missing values in a column, it will give you the frequency counts.

Print the value counts for 'Borough'

```
In [9]: print(df['Borough'].value_counts(dropna=False))

MANHATTAN        6310
BROOKLYN         2866
QUEENS           2121
BRONX             974
STATEN ISLAND     575
Name: Borough, dtype: int64
```

Print the value_counts for 'State'

```
In [10]: print(df['State'].value_counts(dropna=False))

NY     12391
NJ       241
PA        38
CA        20
OH        19
IL        17
FL        17
CT        16
```

Print the value counts for 'Site Fill'

```
In [11]: print(df['Site Fill'].value_counts(dropna=False))

NOT APPLICABLE                                 7806
NaN                                            4205
ON-SITE                                         519
OFF-SITE                                        186
USE UNDER 300 CU.YD                             130
Name: Site Fill, dtype: int64
```

3. Visual exploratory data analysis

3.1 Visualizing single variables with histograms

Until now, you've been looking at descriptive statistics of your data. One of the best ways to confirm what the numbers are telling you is to plot and visualize the data. We will use the Python library **matplotlib** to visualize data.

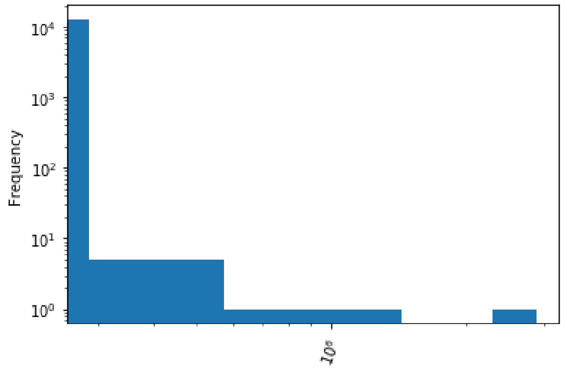
You'll start by visualizing single variables using a histogram for numeric values. The column you will work on in this exercise is 'Existing Zoning Sqft'.

Import matplotlib.pyplot

```
In [12]: import matplotlib.pyplot as plt
```

Plot the histogram

```
In [13]: df['Existing Zoning Sqft'].plot(kind='hist', rot=70, logx=True, logy=True)
         plt.show()
```



3.2 Visualizing multiple variables with boxplots

Histograms are a good way of visualizing single variables. To visualize multiple variables, boxplots are useful, especially when one of the variables is categorical.

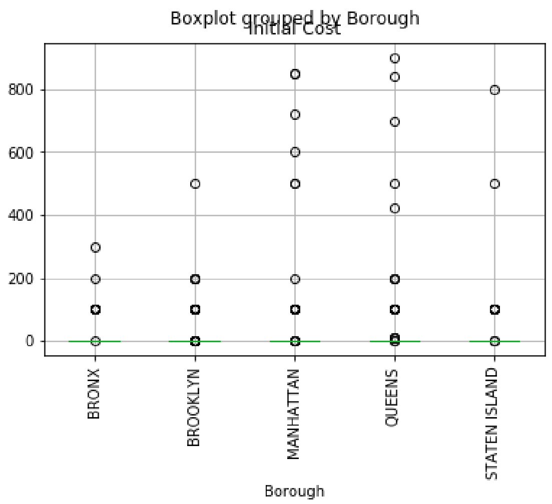
Import necessary modules

```
In [14]: import pandas as pd
         import matplotlib.pyplot as plt
```

Create the boxplot & Display the plot

```
In [15]: df['Initial Cost'].head(5)
Out[15]: 0    75,000
         1         0
         2    30,000
         3     1,500
         4    19,500
         Name: Initial Cost, dtype: object
```

```
In [16]: df['Initial Cost'] = pd.to_numeric(df['Initial Cost'], errors='coerce')
         df.boxplot(column='Initial Cost', by='Borough', rot=90)
         plt.show()
```

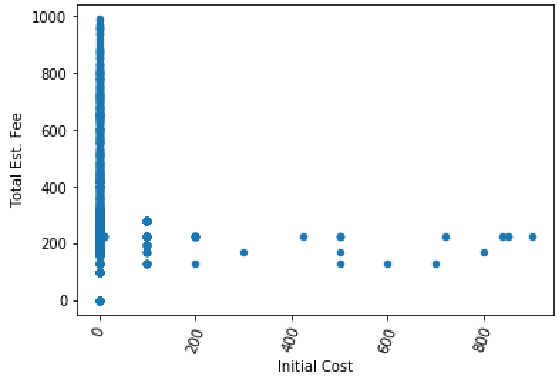


3.3 Visualizing multiple variables with scatter plots

Boxplots are great when you have a numeric column that you want to compare across different categories. When you want to visualize two numeric columns, scatter plots are ideal.

Create and display the first scatter plot

In [18]:
```
df['Total Est. Fee'] = pd.to_numeric(df['Total Est. Fee'], errors='coerce')
df.plot(kind='scatter', x='Initial Cost', y='Total Est. Fee', rot=70)
plt.show()
```



Get acquainted with the dataset now by exploring it with pandas! This initial exploratory analysis is a crucial first step of data cleaning.

II. Cleaning data for analysis

1. Data types

1.1 Converting data types

In [1]:
```
import pandas as pd
```

In [2]:
```
tips = pd.read_csv("tips.csv")
```

In [3]:
```
print(tips.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 8 columns):
total_bill      244 non-null float64
tip             244 non-null float64
sex             244 non-null object
smoker          244 non-null object
day             244 non-null object
time            244 non-null object
size            244 non-null int64
total_dollar    244 non-null object
dtypes: float64(2), int64(1), object(5)
memory usage: 15.3+ KB
None
```

Convert the sex column to type 'category'

In [4]:
```
tips.sex = tips.sex.astype('category')
```

Convert the smoker column to type 'category'

In [5]:
```
tips.smoker = tips.smoker.astype('category')
```

Print the info of tips

```
In [6]: print(tips.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 244 entries, 0 to 243
Data columns (total 8 columns):
total_bill      244 non-null float64
tip             244 non-null float64
sex             244 non-null category
smoker          244 non-null category
day             244 non-null object
time            244 non-null object
size            244 non-null int64
total_dollar    244 non-null object
dtypes: category(2), float64(2), int64(1), object(3)
memory usage: 12.2+ KB
None
```

1.2 Working with numeric data

If you expect the data type of a column to be numeric (*int* or *float*), but instead it is of type *object*, this typically means that there is a non-numeric value in the column, which also signifies bad data.

```
In [7]: df = pd.read_csv('dob_job_application_filings_subset.csv',low_memory=False)
```

```
In [8]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12846 entries, 0 to 12845
Data columns (total 82 columns):
Job #                           12846 non-null int64
Doc #                           12846 non-null int64
Borough                         12846 non-null object
House #                         12846 non-null object
Street Name                     12846 non-null object
Block                           12846 non-null int64
```

Note the "Initial Cost" column, it now has the type of object. It's time to convert 'Initial Cost' to a numeric dtype

```
In [9]: df['Initial Cost'] = pd.to_numeric(df['Initial Cost'], errors='coerce')
```

```
In [10]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 12846 entries, 0 to 12845
Data columns (total 82 columns):
Job #                           12846 non-null int64
Doc #                           12846 non-null int64
Borough                         12846 non-null object
House #                         12846 non-null object
Street Name                     12846 non-null object
Block                           12846 non-null int64
Lot                             12846 non-null int64
Bin #                           12846 non-null int64
```

2. Using regular expressions to clean strings

2.1 String parsing with regular expressions

When working *Strings*, it is sometimes necessary to write a regular expression to look for properly entered values. Phone numbers in a dataset is a common field that needs to be checked for validity. Your job in this exercise is to define a regular expression to match US phone numbers that fit the pattern of xxx-xxx-xxxx.

```
In [11]: # Import the regular expression module
         import re

         # Compile the pattern: prog
         prog = re.compile('\d{3}-\d{3}-\d{4}')

         # See if the pattern matches
         result = prog.match('123-456-7890')
         print(bool(result))

         # See if the pattern matches
         result = prog.match('1123-456-7890')
         print(bool(result))

True
False
```

2.2 Extracting numerical values from strings

Extracting numbers from strings is a common task, particularly when working with unstructured data or log files.

Say you have the following string: *'the recipe calls for 6 strawberries and 2 bananas'*.

It would be useful to extract the 6 and the 2 from this string to be saved for later use when comparing strawberry to banana ratios.

```
In [12]: # Import the regular expression module
         import re

         # Find the numeric values: matches
         matches = re.findall('\d+', 'the recipe calls for 10 strawberries and 1 banana')

         # Print the matches
         print(matches)

['10', '1']
```

2.3 Pattern matching

In this exercise, you'll continue practicing your regular expression skills.

```
In [13]: # Write the first pattern
         pattern1 = bool(re.match(pattern='\d{3}-\d{3}-\d{4}', string='123-456-7890'))
         print(pattern1)

         # Write the second pattern
         pattern2 = bool(re.match(pattern='\$\d*\.\d*', string='$123.45'))
         print(pattern2)

         # Write the third pattern
         pattern3 = bool(re.match(pattern='[A-Z]\w*', string='Australia'))
         print(pattern3)

True
True
True
```

3. Using functions to clean data

3.1 Custom functions to clean data

You'll now practice writing functions to clean data.

The tips dataset has been pre-loaded into a DataFrame called tips. It has a 'sex' column that contains the values 'Male' or 'Female'. Your job is to write a function that will recode 'Male' to 1, 'Female' to 0, and return np.nan for all entries of 'sex' that are neither 'Male' nor 'Female'.

```
In [14]: # Define recode_sex()
def recode_sex(sex_value):

    # Return 1 if sex_value is 'Male'
    if sex_value == 'Male':
        return 1

    # Return 0 if sex_value is 'Female'
    elif sex_value == 'Female':
        return 0

    # Return np.nan
    else:
        return np.nan

# Apply the function to the sex column
tips['sex_recode'] = tips.sex.apply(recode_sex)

# Print the first five rows of tips
print(tips.head())
```

```
   total_bill   tip     sex smoker  day    time  size total_dollar  sex_recode
0       16.99  1.01  Female     No  Sun  Dinner     2       $16.99           0
1       10.34  1.66    Male     No  Sun  Dinner     3       $10.34           1
2       21.01  3.50    Male     No  Sun  Dinner     3       $21.01           1
3       23.68  3.31    Male     No  Sun  Dinner     2       $23.68           1
4       24.59  3.61  Female     No  Sun  Dinner     4       $24.59           0
```

3.2 Lambda functions

You'll now be introduced to a powerful Python feature that will help you clean your data more effectively: lambda functions. Instead of using the *def* syntax that you used in the previous exercise, lambda functions let you make simple, one-line functions.

```
In [15]: # Write the lambda function using replace
tips['total_dollar_replace'] = tips.total_dollar.apply(lambda x: x.replace('$', ''))

# Print the head of tips
print(tips.head())
```

```
   total_bill   tip     sex smoker  day    time  size total_dollar  sex_recode  \
0       16.99  1.01  Female     No  Sun  Dinner     2       $16.99           0
1       10.34  1.66    Male     No  Sun  Dinner     3       $10.34           1
2       21.01  3.50    Male     No  Sun  Dinner     3       $21.01           1
3       23.68  3.31    Male     No  Sun  Dinner     2       $23.68           1
4       24.59  3.61  Female     No  Sun  Dinner     4       $24.59           0

  total_dollar_replace
0                16.99
1                10.34
2                21.01
3                23.68
4                24.59
```

4. Outliers and missing data

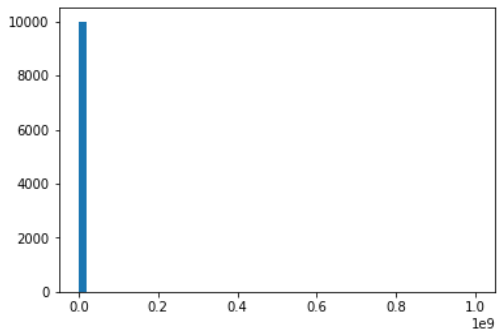
4.1 Outliers

Sometimes outliers can mess up an analysis; you usually don't want a handful of data points to skew the overall results. Let's revisit our example of income data, with Donald Trump thrown in:

```
In [16]: %matplotlib inline
import numpy as np

incomes = np.random.normal(27000, 15000, 10000)
incomes = np.append(incomes, [1000000000])

import matplotlib.pyplot as plt
plt.hist(incomes, 50)
plt.show()
```



That's not very helpful to look at. One billionaire ended up squeezing everybody else into a single line in my histogram. Plus it skewed my mean income significantly:

```
In [17]: incomes.mean()
Out[17]: 127007.64410448549
```

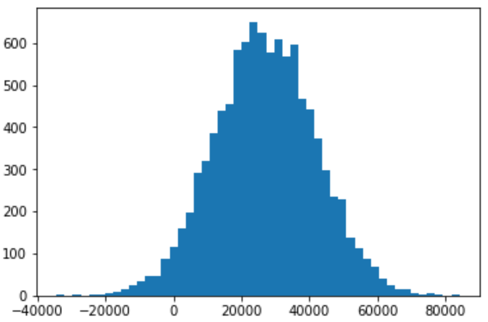
It's important to dig into what is causing your outliers, and understand where they are coming from. You also need to think about whether removing them is a valid thing to do, given the spirit of what it is you're trying to analyze. If I know I want to understand more about the incomes of "typical Americans", filtering out billionaires seems like a legitimate thing to do.

Here's something a little more robust than filtering out billionaires - it filters out anything beyond two standard deviations of the median value in the data set:

```
In [18]: def reject_outliers(data):
             u = np.median(data)
             s = np.std(data)
             filtered = [e for e in data if (u - 2 * s < e < u + 2 * s)]
             return filtered

         filtered = reject_outliers(incomes)

         plt.hist(filtered, 50)
         plt.show()
```



That looks better. And, our mean is more, well, meangingful now as well:

```
In [19]: np.mean(filtered)
Out[19]: 27020.344868895947
```

4.2 Activity: Outliers

Instead of a single outlier, add several randomly-generated outliers to the data. Experiment with different values of the multiple of the standard deviation to identify outliers, and see what effect it has on the final results.

4.3 Filling missing data

It's rare to have a (real-world) dataset without any missing values, and it's important to deal with them because certain calculations cannot handle missing values while some calculations will, by default, skip over any missing values.

Also, understanding how much missing data you have, and thinking about where it comes from is crucial to making unbiased interpretations of data.

```
In [20]: # Load air quality
airquality = pd.read_csv('airquality.csv')

# Calculate the mean of the Ozone column: oz_mean
oz_mean = airquality.Ozone.mean()

# Replace all the missing values in the Ozone column with the mean
airquality['Ozone'] = airquality.Ozone.fillna(oz_mean)

# Print the info of airquality
print(airquality.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 153 entries, 0 to 152
Data columns (total 6 columns):
Ozone      153 non-null float64
Solar.R    146 non-null float64
Wind       153 non-null float64
Temp       153 non-null int64
Month      153 non-null int64
Day        153 non-null int64
dtypes: float64(3), int64(3)
memory usage: 7.2 KB
None
```

III. Dimensionality Reduction

1. Principal Component Analysis

PCA is a dimensionality reduction technique; it lets you distill multi-dimensional data down to fewer dimensions, selecting new dimensions that preserve variance in the data as best it can.

```
In [68]: import matplotlib.pyplot as plt
import pandas as pd

from sklearn.decomposition import PCA as sklearnPCA
```

```
In [69]: url = 'https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data'
data = pd.read_csv(url,header=None)

y = data[4] # Split off classifications
X = data.iloc[:,0:4] # Split off features
```

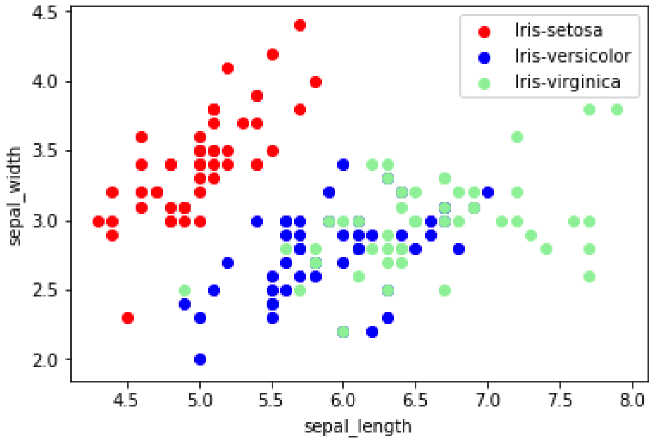
So, this tells us our data set has 150 samples (individual flowers) in it. It has 4 dimensions - called features here, and three distinct Iris species that each flower is classified into.

While we can visualize 2 or even 3 dimensions of data pretty easily, visualizing 4D data isn't something our brains can do. So let's distill this down to 2 dimensions, and see how well it works. A simple approach to visualizing multi-dimensional data is to select two (or three) dimensions and plot the data as seen in that plane. For example, I could plot the sepal_length vs. sepal_width plane as a two-dimensional “slice” of the original dataset:

```
In [75]: # three different scatter series so the class labels in the legend are distinct
plt.scatter(X[y=='Iris-setosa'].iloc[:,0], X[y=='Iris-setosa'].iloc[:,1], label='Iris-setosa', c='red')
plt.scatter(X[y=='Iris-versicolor'].iloc[:,0], X[y=='Iris-versicolor'].iloc[:,1], label='Iris-versicolor', c='blue')
plt.scatter(X[y=='Iris-virginica'].iloc[:,0], X[y=='Iris-virginica'].iloc[:,1], label='Iris-virginica', c='lightgreen')

# Prettify the graph
plt.legend()
plt.xlabel('sepal_length')
plt.ylabel('sepal_width')

# display
plt.show()
```



Before we go further, we should apply feature scaling to our dataset. In this example, I will simply rescale the data to a [0,1] range, but it is also common to standardize the data to have a zero mean and unit standard deviation:

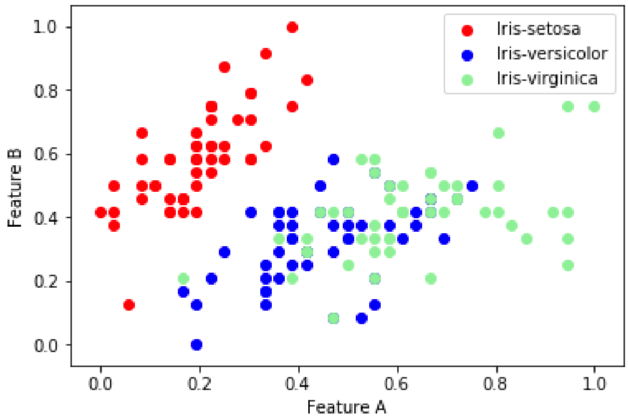
In [71]:
```
X_norm = (X - X.min())/(X.max() - X.min())
```

Plot the X_norm again:

In [72]:
```
# three different scatter series so the class labels in the legend are distinct
plt.scatter(X_norm[y=='Iris-setosa'].iloc[:,0], X_norm[y=='Iris-setosa'].iloc[:,1], label='Iris-setosa', c='red')
plt.scatter(X_norm[y=='Iris-versicolor'].iloc[:,0], X_norm[y=='Iris-versicolor'].iloc[:,1], label='Iris-versicolor', c='blue')
plt.scatter(X_norm[y=='Iris-virginica'].iloc[:,0], X_norm[y=='Iris-virginica'].iloc[:,1], label='Iris-virginica', c='lightgreen')

# Prettify the graph
plt.legend()
plt.xlabel('Feature A')
plt.ylabel('Feature B')

# display
plt.show()
```



In Python, we can use PCA by first fitting an sklearn PCA object to the normalized dataset, then looking at the transformed matrix.

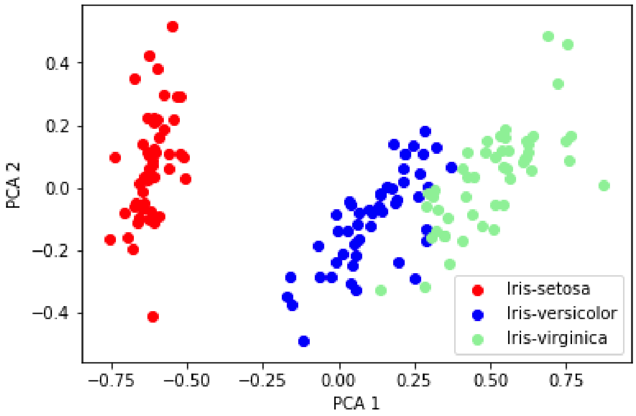
In [73]:
```
pca = sklearnPCA(n_components=2) #2-dimensional PCA
transformed = pd.DataFrame(pca.fit_transform(X_norm))
```

As promised, now that we have a 2D representation of our data, we can plot it:

In [74]:
```
# three different scatter series so the class labels in the legend are distinct
plt.scatter(transformed[y=='Iris-setosa'].iloc[:,0], transformed[y=='Iris-setosa'].iloc[:,1], label='Iris-setosa', c='red')
plt.scatter(transformed[y=='Iris-versicolor'].iloc[:,0],
            transformed[y=='Iris-versicolor'].iloc[:,1], label='Iris-versicolor', c='blue')
plt.scatter(transformed[y=='Iris-virginica'].iloc[:,0],
            transformed[y=='Iris-virginica'].iloc[:,1], label='Iris-virginica', c='lightgreen')

# Prettify the graph
plt.legend()
plt.xlabel('PCA 1')
plt.ylabel('PCA 2')

# display
plt.show()
```



You can see the three different types of Iris are still clustered pretty well.

2. Activity

Please try to do a PCA down to 2 components, and measure the results with this sample data: https://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data.