# 2. Multivariate Data Analysis

**Aim:** to carry out some simple multivariate analyses, with a focus on principal components analysis (PCA) and linear discriminant analysis (LDA)

**Description:**

Univariate, bivariate, and multivariate analysis are three common approaches used in statistics and data analysis to explore and understand data.

1. Univariate Analysis: —Definition: Univariate analysis involves the examination of a single variable (or feature) at a time. It is the simplest form of data analysis and is often used to describe the characteristics of individual variables.

—Purpose: Univariate analysis is primarily used to summarize and visualize the distribution of a single variable, assess its central tendency (mean, median, mode), dispersion (range, variance, standard deviation), and shape (e.g., histogram, box plot). It helps in understanding the characteristics of a single variable in isolation.

2. Bivariate Analysis: —Definition: Bivariate analysis involves the analysis of two variables simultaneously to determine if there is a relationship or association between them. It explores how changes in one variable are related to changes in another.

—Purpose: Bivariate analysis is used to understand the relationship, correlation, or association between two variables. Common techniques for bivariate analysis include scatter plots, correlation coefficients (e.g., Pearson's correlation), and contingency tables (for categorical variables). It helps answer questions like, "Is there a relationship between a person's age and their income?"

3. Multivariate Analysis: —Definition: Multivariate analysis involves the simultaneous analysis of three or more variables to understand the complex relationships and interactions among them. It extends beyond the scope of univariate and bivariate analysis to consider multiple factors at once.

—Purpose: Multivariate analysis is used when there are multiple variables at play, and researchers want to explore how these variables interact and influence each other. It encompasses various techniques, including multiple regression, principal component analysis (PCA), factor analysis, cluster analysis, and multivariate analysis of variance (MANOVA). Multivariate analysis is essential for understanding more complex relationships in data, such as predicting an outcome based on multiple predictor variables or clustering similar observations based on multiple characteristics.

In summary:

- Univariate analysis focuses on understanding individual variables.

- Bivariate analysis examines relationships between two variables.

- Multivariate analysis deals with the interactions and relationships among three or more variables.

### Libraries

The following Python libraries are used here:

- pandas: The Python Data Analysis Library is used for storing the data in dataframes and manipulation.
- numpy: Python scientific computing library.
- matplotlib: Python plotting library.
- seaborn: Statistical data visualization based on matplotlib.
- scikit-learn: Sklearn is a machine learning library for Python.
- scipy.stats: Provides a number of probability distributions and statistical functions.

These should have been installed for you if you have installed the Anaconda Python distribution.

This code was created with python 2.7 version. For exact details, including versions of the other libraries, see the PDF.

### Importing the libraries

```
from pydoc import help
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.preprocessing import scale
from sklearn.decomposition import PCA
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from scipy import stats
from IPython.display import display, HTML

# figures inline in notebook
%matplotlib inline

np.set_printoptions(suppress=True)

DISPLAY_MAX_ROWS = 20  # number of max rows to print for a DataFrame
pd.set_option('display.max_rows', DISPLAY_MAX_ROWS)
```

## Reading Multivariate Analysis Data into Python

The first thing that you will want to do to analyse your multivariate data will be to read it into Python, and to plot the data. For data analysis an I will be using the Python Data Analysis Library (pandas, imported as pd), which provides a number of useful functions for reading and analyzing the data, as well as a DataFrame storage structure.

For example, the file http://archive.ics.uci.edu/ml/machine-learning-databases/wine/wine.data contains data on concentrations of 13 different chemicals in wines grown in the same region in Italy that are derived from three different cultivars. The data set looks like this:

1,14.23,1.71,2.43,15.6,127,2.8,3.06,.28,2.29,5.64,1.04,3.92,1065

```
1,13.2,1.78,2.14,11.2,100,2.65,2.76,.26,1.28,4.38,1.05,3.4,1050
1,13.16,2.36,2.67,18.6,101,2.8,3.24,.3,2.81,5.68,1.03,3.17,1185
1,14.37,1.95,2.5,16.8,113,3.85,3.49,.24,2.18,7.8,.86,3.45,1480
1,13.24,2.59,2.87,21,118,2.8,2.69,.39,1.82,4.32,1.04,2.93,735
...
```

There is one row per wine sample. The first column contains the cultivar of a wine sample (labelled 1, 2 or 3), and the following thirteen columns contain the concentrations of the 13 different chemicals in that sample. The columns are separated by commas, i.e. it is a comma-separated (csv) file without a header row.

The data can be read in a pandas dataframe using the read_csv() function. The argument header=None tells the function that there is no header in the beginning of the file.

```
data = pd.read_csv("http://archive.ics.uci.edu/ml/machine-learning-
databases/wine/wine.data", header=None)
data.columns = ["V"+str(i) for i in range(1, len(data.columns)+1)]  # rename
column names to be similar to R naming convention
data.V1 = data.V1.astype(str)
X = data.loc[:, "V2":]  # independent variables data
y = data.V1  # dependednt variable data
data
```

In this case the data on 178 samples of wine has been read into the variable data.

## Plotting Multivariate Data

Once you have read a multivariate data set into python, the next step is usually to make a plot of the data.

### A Matrix Scatterplot

One common way of plotting multivariate data is to make a *matrix scatterplot*, showing each pair of variables plotted against each other. We can use the `scatter_matrix()` function from the `pandas.tools.plotting` package to do this.

To use the `scatter_matrix()` function, you need to give it as its input the variables that you want included in the plot. Say for example, that we just want to include the variables corresponding to the concentrations of the first five chemicals. These are stored in columns V2-V6 of the variable `data`. The parameter `diagonal` allows us to specify whether to plot a histogram (`"hist"`) or a Kernel Density Estimation (`"kde"`) for the variable. We can extract just these columns from the variable `data` by typing:

```
data.loc[:, "V2":"V6"]
```

To make a matrix scatterplot of just these 5 variables using the `scatter_matrix()` function we type:

```
pd.tools.plotting.scatter_matrix(data.loc[:, "V2":"V6"], diagonal="kde")
plt.tight_layout()
plt.show()
```

In this matrix scatterplot, the diagonal cells show histograms of each of the variables, in this case the concentrations of the first five chemicals (variables V2, V3, V4, V5, V6).

Each of the off-diagonal cells is a scatterplot of two of the five chemicals, for example, the second cell in the first row is a scatterplot of V2 (y-axis) against V3 (x-axis).

### A Profile Plot

Another type of plot that is useful is a *profile plot*, which shows the variation in each of the variables, by plotting the value of each of the variables for each of the samples.

This can be achieved using `pandas` plot facilities, which are built upon `matplotlib`, by running the following:

```
ax = data[["V2","V3","V4","V5","V6"]].plot()
ax.legend(loc='center left', bbox_to_anchor=(1, 0.5));
```

It is clear from the profile plot that the mean and standard deviation for V6 is quite a lot higher than that for the other variables.

## Calculating Summary Statistics for Multivariate Data

Another thing that you are likely to want to do is to calculate summary statistics such as the mean and standard deviation for each of the variables in your multivariate data set.

This is easy to do, using the `mean()` and `std()` functions in `numpy` and applying them to the dataframe using its member function `apply`.

Pandas allows to do simple operations directly calling them as methods, for example we could do compute the means of a dataframe `df` by calling `df.mean()`.

or example, say we want to calculate the mean and standard deviations of each of the 13 chemical concentrations in the wine samples. These are stored in columns V2-V14 of the variable `data`, which has been previously assigned to `X` for convenience. So we type:

```
X.apply(np.mean)
```

This tells us that the mean of variable V2 is 13.000618, the mean of V3 is 2.336348, and so on.

Similarly, to get the standard deviations of the 13 chemical concentrations, we type:

```
X.apply(np.std)
```

### Means and Variances Per Group

It is often interesting to calculate the means and standard deviations for just the samples from a particular group, for example, for the wine samples from each cultivar. The cultivar is stored in the column V1 of the variable `data`, which has been previously assigned to `y` for convenience.

To extract out the data for just cultivar 2, we can type:

```
class2data = data[y=="2"]
```

We can then calculate the mean and standard deviations of the 13 chemicals' concentrations, for just the cultivar 2 samples:

```
class2data.loc[:, "V2":].apply(np.mean)

class2data.loc[:, "V2":].apply(np.std)
```

However, for convenience, you might want to use the function `printMeanAndSdByGroup()` below, which prints out the mean and standard deviation of the variables for each group in your data set:

```
def printMeanAndSdByGroup(variables, groupvariable):
    data_groupby = variables.groupby(groupvariable)
    print("## Means:")
    display(data_groupby.apply(np.mean))
    print("\n## Standard deviations:")
    display(data_groupby.apply(np.std))
    print("\n## Sample sizes:")
    display(pd.DataFrame(data_groupby.apply(len)))
```

The arguments of the function are the variables that you want to calculate means and standard deviations for (`X`), and the variable containing the group of each sample (`y`). For example, to calculate the mean and standard deviation for each of the 13 chemical concentrations, for each of the three different wine cultivars, we type:

```
printMeanAndSdByGroup(X, y)
```

The function `printMeanAndSdByGroup()` also prints out the number of samples in each group. In this case, we see that there are 59 samples of cultivar 1, 71 of cultivar 2, and 48 of cultivar 3.

## Calculating Correlations for Multivariate Data

It is often of interest to investigate whether any of the variables in a multivariate data set are significantly correlated.

To calculate the linear (Pearson) correlation coefficient for a pair of variables, you can use the `pearsonr()` function from `scipy.stats` package. For example, to calculate the correlation coefficient for the first two chemicals' concentrations, V2 and V3, we type:

```
corr = stats.pearsonr(X.V2, X.V3)
print("p-value:\t", corr[1])
print("cor:\t\t", corr[0])

p-value:      0.210081985971
cor:          0.0943969409104
```

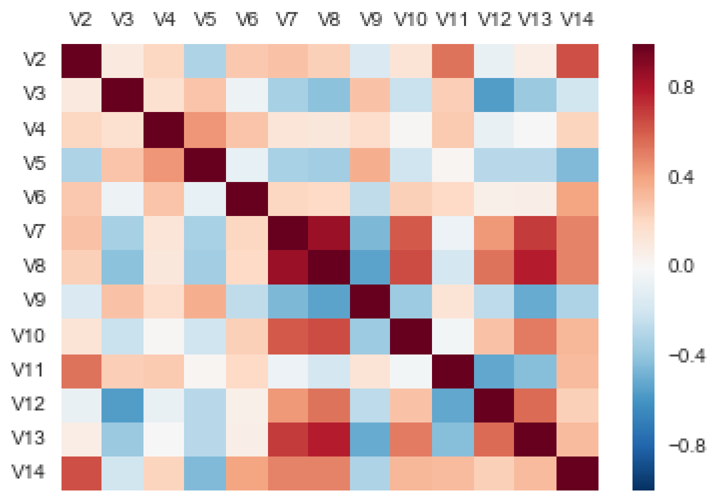
This tells us that the correlation coefficient is about 0.094, which is a very weak correlation. Furthermore, the *p-value* for the statistical test of whether the correlation coefficient is significantly different from zero is 0.21. This is much greater than 0.05 (which we can use here as a cutoff for statistical significance), so there is very weak evidence that that the correlation is non-zero.

If you have a lot of variables, you can use the `pandas.DataFrame` method `corr()` to calculate a correlation matrix that shows the correlation coefficient for each pair of variables.

```
corrmat = X.corr()
corrmat
```

A better graphical representation of the correlation matrix is via a correlation matrix plot in the form of a *heatmap*.

```
sns.heatmap(corrmat, vmax=1., square=False).xaxis.tick_top()
```



Although the correlation matrix and diagrams are useful for quickly looking to identify the strongest correlations, they still require labor work to find the top `N` strongest correlations. For this you can use the function `mosthighlycorrelated()` below.

The function `mosthighlycorrelated()` will print out the linear correlation coefficients for each pair of variables in your data set, in order of the correlation coefficient. This lets you see very easily which pair of variables are most highly correlated.

```
def mosthighlycorrelated(mydataframe, numtoreport):
    # find the correlations
    cormatrix = mydataframe.corr()
    # set the correlations on the diagonal or lower triangle to zero,
    # so they will not be reported as the highest ones:
    cormatrix *= np.tri(*cormatrix.values.shape, k=-1).T
    # find the top n correlations
    cormatrix = cormatrix.stack()
    cormatrix = 
cormatrix.reindex(cormatrix.abs().sort_values(ascending=False).index).reset_inde
x()
    # assign human-friendly names
    cormatrix.columns = ["FirstVariable", "SecondVariable", "Correlation"]
    return cormatrix.head(numtoreport)
```

The arguments of the function are the variables that you want to calculate the correlations for, and the number of top correlation coefficients to print out (for example, you can tell it to print out the largest 10 correlation coefficients, or the largest 20).

For example, to calculate correlation coefficients between the concentrations of the 13 chemicals in the wine samples, and to print out the top 10 pairwise correlation coefficients, you can type:

```
mosthighlycorrelated(X, 10)
```

## Principal Component Analysis (PCA)

The purpose of principal component analysis is to find the best low-dimensional representation of the variation in a multivariate data set. For example, in the case of the wine data set, we have 13 chemical concentrations describing wine samples from three different cultivars. We can carry out a principal component analysis to investigate whether we can capture most of the variation between samples using a smaller number of new variables (principal components), where each of these new variables is a linear combination of all or some of the 13 chemical concentrations.

To carry out a principal component analysis (PCA) on a multivariate data set, the first step is often to standardise the variables under study using the `scale()` function (see above). This is necessary if the input variables have very different variances, which is true in this case as the concentrations of the 13 chemicals have very different variances (see above).

Once you have standardised your variables, you can carry out a principal component analysis using the `PCA` class from `sklearn.decomposition` package and its `fit` method, which fits the model with the data `X`. The default `solver` is Singular Value Decomposition (“svd”). For more information you can type `help(PCA)` in the python console.

For example, to standardise the concentrations of the 13 chemicals in the wine samples, and carry out a principal components analysis on the standardised concentrations, we type:

```
pca = PCA().fit(standardisedX)
```

You can get a summary of the principal component analysis results using the `pca_summary()` function below, which simulates the output of R’s `summary` function on a PCA model:

```
def pca_summary(pca, standardised_data, out=True):
    names = ["PC"+str(i) for i in range(1, len(pca.explained_variance_ratio_)
+1)]
    a = list(np.std(pca.transform(standardised_data), axis=0))
    b = list(pca.explained_variance_ratio_)
    c = [np.sum(pca.explained_variance_ratio_[:i]) for i in range(1,
len(pca.explained_variance_ratio_)+1)]
    columns = pd.MultiIndex.from_tuples([("sdev", "Standard deviation"),
("varprop", "Proportion of Variance"), ("cumprop", "Cumulative Proportion")])
    summary = pd.DataFrame(zip(a, b, c), index=names, columns=columns)
    if out:
        print("Importance of components:")
        display(summary)
    return summary
```

The parameters of the `print_pca_summary` function are:

- `pca`: A PCA object
- `standardised_data`: The standardised data
- `out (True)`: Print to standard output

```
summary = pca_summary(pca, standardisedX)
```

```
summary.sdev
```

The total variance explained by the components is the sum of the variances of the components:

```
np.sum(summary.sdev**2)
```

```
Standard deviation    13
dtype: float64
```

n this case, we see that the total variance is 13, which is equal to the number of standardised variables (13 variables). This is because for standardised data, the variance of each standardised variable is 1.

### Deciding How Many Principal Components to Retain

Another way of deciding how many components to retain is to use *Kaiser's criterion*: that we should only retain principal components for which the variance is above 1 (when principal component analysis was applied to standardised data). We can check this by finding the variance of each of the principal components:

```
summary.sdev**2
```

We see that the variance is above 1 for principal components 1, 2, and 3 (which have variances 4.71, 2.50, and 1.45, respectively). Therefore, using Kaiser's criterion, we would retain the first three principal components.

## Linear Discriminant Analysis (LDA)

The purpose of principal component analysis is to find the best low-dimensional representation of the variation in a multivariate data set. For example, in the wine data set, we have 13 chemical concentrations describing wine samples from three cultivars.

The purpose of linear discriminant analysis (LDA) is to find the linear combinations of the original variables (the 13 chemical concentrations here) that gives the best possible separation between the groups (wine cultivars here) in our data set. *Linear discriminant analysis* is also known as *canonical discriminant analysis*, or simply *discriminant analysis*.

If we want to separate the wines by cultivar, the wines come from three different cultivars, so the number of groups (G) is 3, and the number of variables is 13 (13 chemicals' concentrations; p = 13). The maximum number of useful discriminant functions that can separate the wines by cultivar is the minimum of G-1 and p, and so in this case it is the minimum of 2 and 13, which is 2. Thus, we can find at most 2 useful discriminant functions to separate the wines by cultivar, using the 13 chemical concentration variables.

You can carry out a linear discriminant analysis by using the `LinearDiscriminantAnalysis` class model from the module `sklearn.discriminant_analysis` and using its method `fit()` to fit our `X, y` data.

For example, to carry out a linear discriminant analysis using the 13 chemical concentrations in the wine samples, we type:

```
lda = LinearDiscriminantAnalysis().fit(X, y)
```

## Loadings for the Discriminant Functions

The values of the loadings of the discriminant functions for the wine data are stored in the `scalings_` member of the `lda` object model. For a pretty print we can type:

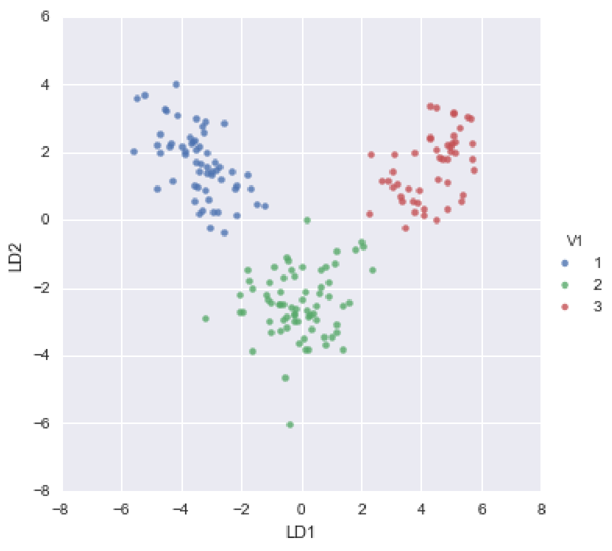
```
def pretty_scalings(lda, X, out=False):
    ret = pd.DataFrame(lda.scalings_, index=X.columns, columns=["LD"+str(i+1)
for i in range(lda.scalings_.shape[1])])
    if out:
        print("Coefficients of linear discriminants:")
        display(ret)
    return ret

pretty_scalings_ = pretty_scalings(lda, X, out=True)
```

## Scatterplots of the Discriminant Functions

We can obtain a scatterplot of the best two discriminant functions, with the data points labelled by cultivar, by typing:

```
sns.lmplot("LD1", "LD2", lda_values["x"].join(y), hue="V1", fit_reg=False);
```



For More learning, see the links and PDF, which can be found at the link given below:

[https://python-for-multivariate-analysis.readthedocs.io/a_little_book_of_python_for_multivariate_analysis.html](https://python-for-multivariate-analysis.readthedocs.io/a_little_book_of_python_for_multivariate_analysis.html)

[https://python-for-multivariate-analysis.readthedocs.io/_/downloads/en/latest/pdf/](https://python-for-multivariate-analysis.readthedocs.io/_/downloads/en/latest/pdf/)