A Little Book of Python for Multivariate Analysis Documentation

Release 0.1

Yiannis Gatsoulis

February 21, 2016

Contents

1 Notes 3

2 Contents 5
- 2.1 A Little Book of Python for Multivariate Analysis . . . 5
  - 2.1.1 Setting up the python environment . . . 5
    - Install Python . . . 5
    - Libraries . . . 5
    - Importing the libraries . . . 6
    - Python console . . . 7
  - 2.1.2 Reading Multivariate Analysis Data into Python . . . 7
  - 2.1.3 Plotting Multivariate Data . . . 7
    - A Matrix Scatterplot . . . 7
    - A Scatterplot with the Data Points Labelled by their Group . . . 8
    - A Profile Plot . . . 9
  - 2.1.4 Calculating Summary Statistics for Multivariate Data . . . 10
    - Means and Variances Per Group . . . 11
    - Between-groups Variance and Within-groups Variance for a Variable . . . 12
    - Between-groups Covariance and Within-groups Covariance for Two Variables . . . 14
    - Calculating Correlations for Multivariate Data¶[. . . 16
    - Standardising Variables . . . 18
  - 2.1.5 Principal Component Analysis . . . 19
    - Deciding How Many Principal Components to Retain . . . 20
    - Loadings for the Principal Components . . . 21
    - Scatterplots of the Principal Components . . . 24
  - 2.1.6 Linear Discriminant Analysis . . . 26
    - Loadings for the Discriminant Functions . . . 26
    - Separation Achieved by the Discriminant Functions . . . 32
    - A Stacked Histogram of the LDA Values . . . 34
    - Scatterplots of the Discriminant Functions . . . 36
    - Allocation Rules and Misclassification Rate . . . 37
  - 2.1.7 Links and Further Reading . . . 40
  - 2.1.8 Acknowledgements . . . 41
  - 2.1.9 Contact . . . 41
  - 2.1.10 License . . . 41

3 License 43

This booklet tells you how to use the Python ecosystem to carry out some simple multivariate analyses, with a focus on principal components analysis (PCA) and linear discriminant analysis (LDA).

The jupyter notebook can be found on its github repository.

CHAPTER 1

Notes

This booklet assumes that the reader has some basic knowledge of multivariate analyses, and the principal focus of the booklet is not to explain multivariate analyses, but rather to explain how to carry out these analyses using Python.

The naming conventions in the functions were kept like in the original source for compliance. The variables were renamed to more generic names, so it would be possible to load your own dataset and run the notebook as it is if the first column of your data contains the data classes. See the cell that does the reading of the data.

The Python code was aimed to be easy to understand, like the R code in the original source, rather than be computationally and memory efficient.

CHAPTER 2

Contents

2.1 A Little Book of Python for Multivariate Analysis

This booklet tells you how to use the Python ecosystem to carry out some simple multivariate analyses, with a focus on principal components analysis (PCA) and linear discriminant analysis (LDA).

This booklet assumes that the reader has some basic knowledge of multivariate analyses, and the principal focus of the booklet is not to explain multivariate analyses, but rather to explain how to carry out these analyses using Python.

If you are new to multivariate analysis, and want to learn more about any of the concepts presented here, there are a number of good resources, such as for example Multivariate Data Analysis by Hair et. al. or Applied Multivariate Data Analysis by Everitt and Dunn.

In the examples in this booklet, I will be using data sets from the UCI Machine Learning Repository.

2.1.1 Setting up the python environment

Install Python

Although there are a number of ways of getting Python to your system, for a hassle free install and quick start using, I highly recommend downloading and installing Anaconda by Continuum, which is a Python distribution that contains the core packages plus a large number of packages for scientific computing and tools to easily update them, install new ones, create virtual environments, and provide IDEs such as this one, the Jupyter notebook (formerly known as ipython notebook).

This notebook was created with python 2.7 version. For exact details, including versions of the other libraries, see the `%watermark` directive below.

Libraries

Python can typically do less out of the box than other languages, and this is due to being a genaral programming language taking a more modular approach, relying on other packages for specialized tasks.

The following 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.

The libraries versions are:

```
from __future__ import print_function, division  # for compatibility with python 3.x
import warnings
warnings.filterwarnings('ignore')  # don't print out warnings

%install_ext https://raw.githubusercontent.com/rasbt/watermark/master/watermark.py
%load_ext watermark
%watermark -v -m -p python,pandas,numpy,matplotlib,seaborn,scikit-learn,scipy -g
```

```
Installed watermark.py. To use it, type:
  %load_ext watermark
CPython 2.7.11
IPython 4.0.3

python 2.7.11
pandas 0.17.1
numpy 1.10.4
matplotlib 1.5.1
seaborn 0.7.0
scikit-learn 0.17
scipy 0.17.0

compiler   : GCC 4.2.1 (Apple Inc. build 5577)
system     : Darwin
release    : 13.4.0
machine    : x86_64
processor  : i386
CPU cores  : 4
interpreter: 64bit
Git hash   : b584574b9a5080bac2e592d4432f9c17c1845c18
```

Importing the libraries

```
from pydoc import help  # can type in the python console `help(name of function)` to get the document
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)
```

Python console

A useful tool to have aside a notebook for quick experimentation and data visualization is a python console attached. Uncomment the following line if you wish to have one.

```
# %qtconsole
```

2.1.2 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, similar to that found in other popular data analytics languages, such as R.

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=
data.columns = ["V"+str(i) for i in range(1, len(data.columns)+1)]  # rename column names to be simi
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`.

2.1.3 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.

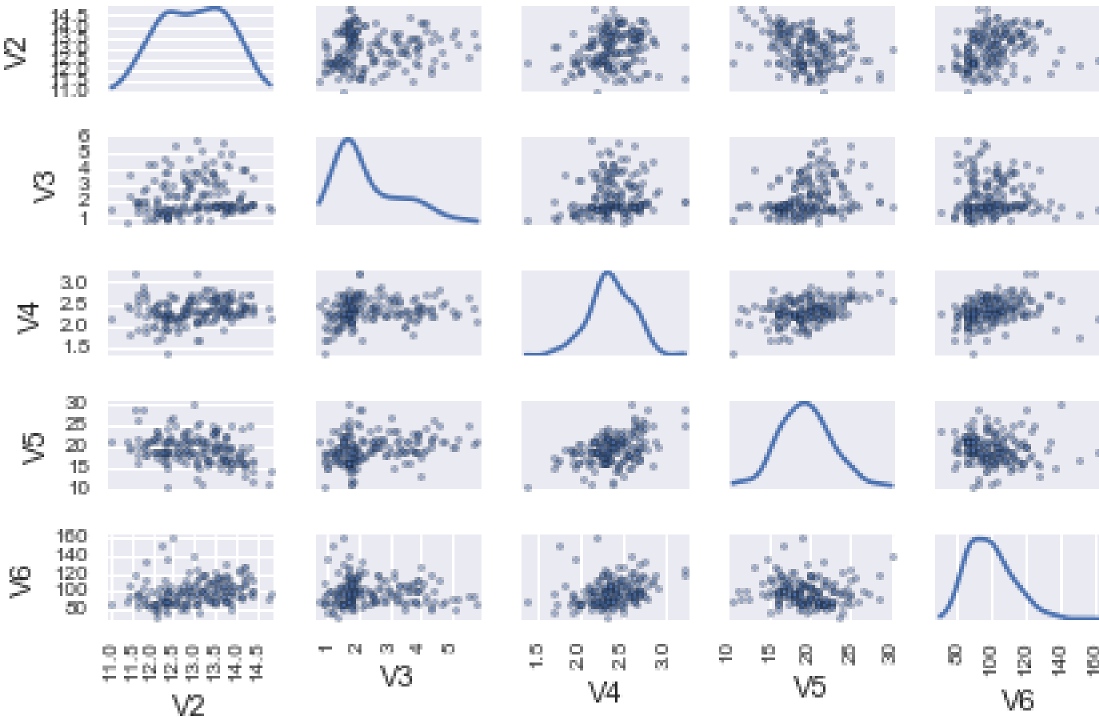
To use the `scatter_matrix()` function, you need to give it as its input the variables that you want included in the plot. Say for example, that we just want to include the variables corresponding to the concentrations of the first five chemicals. These are stored in columns V2-V6 of the variable `data`. 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 Scatterplot with the Data Points Labelled by their Group

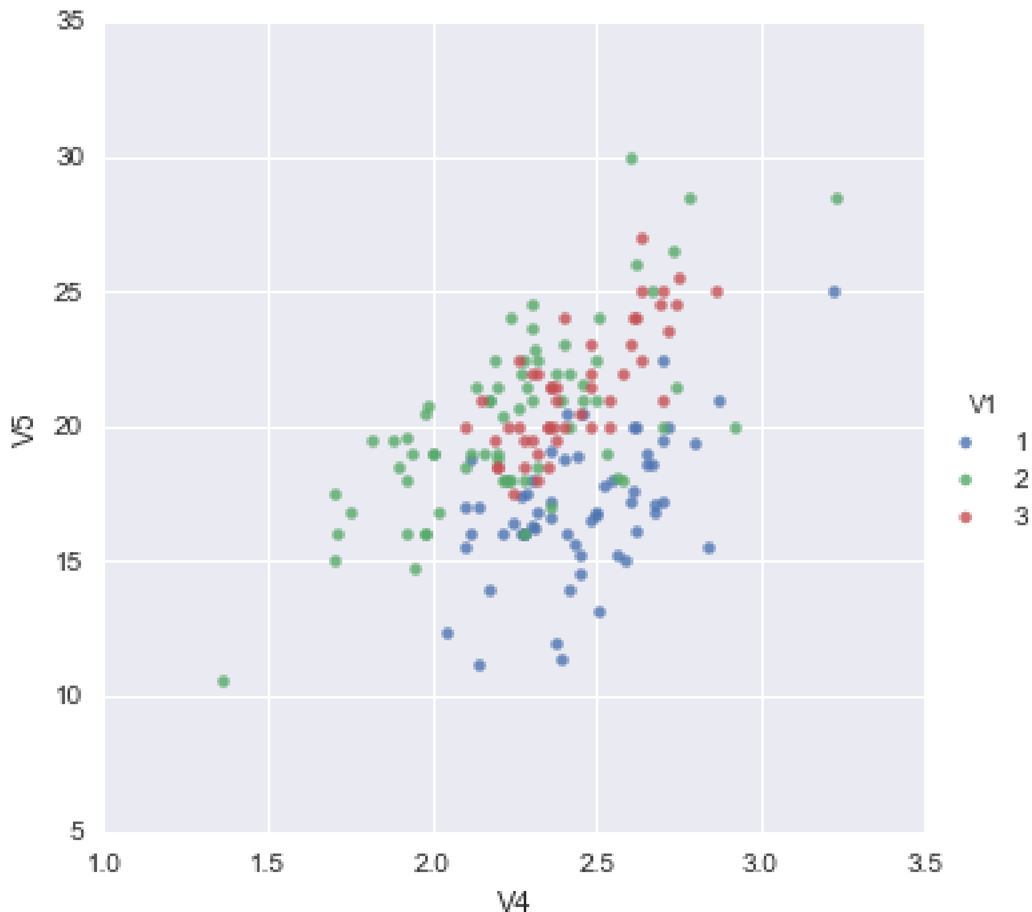
If you see an interesting scatterplot for two variables in the matrix scatterplot, you may want to plot that scatterplot in more detail, with the data points labelled by their group (their cultivar in this case).

For example, in the matrix scatterplot above, the cell in the third column of the fourth row down is a scatterplot of V5 (x-axis) against V4 (y-axis). If you look at this scatterplot, it appears that there may be a positive relationship between V5 and V4.

We may therefore decide to examine the relationship between V5 and V4 more closely, by plotting a scatterplot of these two variables, with the data points labelled by their group (their cultivar). To plot a scatterplot of two variables, we can use the lmplot function from the seaborn package. The V4 and V5 variables are stored in the columns V4 and V5 of the variable data. The first two parameters in the lmplot() function are the columns to be plotted against each other in x-y, the third parameter specifies the data, the hue parameter is the column name used for the

labels of the datapoints, i.e. the classes they belong to, lastly, the `fit_reg` parameter is set to `False` when we do not want to plot a regression model relating to the x-y variables. Therefore, to plot the scatterplot, we type:

```
sns.lmplot("V4", "V5", data, hue="V1", fit_reg=False);
```



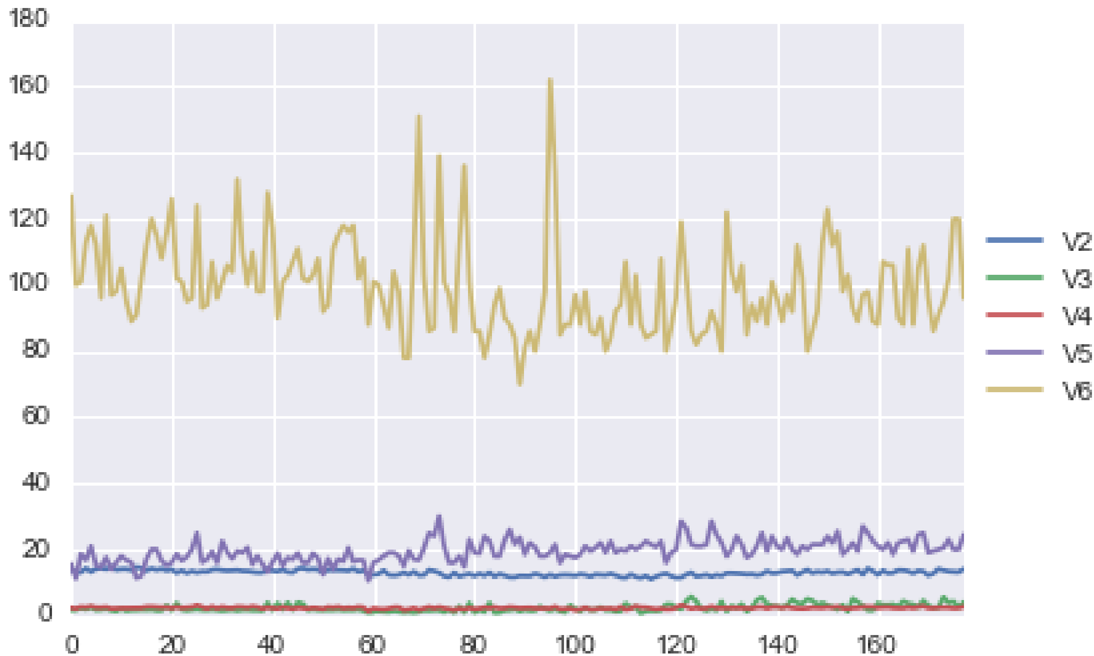
We can see from the scatterplot of V4 versus V5 that the wines from cultivar 2 seem to have lower values of V4 compared to the wines of cultivar 1.

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.

2.1.4 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`.

An alternative option is to use the `apply` method of the `pandas.DataFrame` class, which applies the passed argument function along the input axis of the DataFrame. This method is powerful as it allows passing any function we want to be applied in our data.

For 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)
```

```
V2      13.000618
V3       2.336348
V4       2.366517
V5      19.494944
V6      99.741573
V7       2.295112
V8       2.029270
V9       0.361854
V10      1.590899
V11      5.058090
V12      0.957449
V13      2.611685
```

```
V14    746.893258
dtype: float64
```

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)
```

```
V2       0.809543
V3       1.114004
V4       0.273572
V5       3.330170
V6      14.242308
V7       0.624091
V8       0.996049
V9       0.124103
V10      0.570749
V11      2.311765
V12      0.227929
V13      0.707993
V14    314.021657
dtype: float64
```

We can see here that it would make sense to standardise in order to compare the variables because the variables have very different standard deviations - the standard deviation of V14 is 314.021657, while the standard deviation of V9 is just 0.124103. Thus, in order to compare the variables, we need to standardise each variable so that it has a sample variance of 1 and sample mean of 0. We will explain below how to standardise the variables.

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)
```

```
V2      12.278732
V3       1.932676
V4       2.244789
V5      20.238028
V6      94.549296
V7       2.258873
V8       2.080845
V9       0.363662
V10      1.630282
V11      3.086620
V12      1.056282
V13      2.785352
V14    519.507042
dtype: float64
```

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

```
V2       0.534162
V3       1.008391
V4       0.313238
V5       3.326097
V6      16.635097
V7       0.541507
V8       0.700713
V9       0.123085
V10      0.597813
V11      0.918393
V12      0.201503
V13      0.493064
V14    156.100173
dtype: float64
```

You can calculate the mean and standard deviation of the 13 chemicals' concentrations for just cultivar 1 samples, or for just cultivar 3 samples, in a similar way.

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)
```

```
## Means:
```

```
## Standard deviations:
```

```
## Sample sizes:
```

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.

Between-groups Variance and Within-groups Variance for a Variable

If we want to calculate the within-groups variance for a particular variable (for example, for a particular chemical's concentration), we can use the function `calcWithinGroupsVariance()` below:

```
def calcWithinGroupsVariance(variable, groupvariable):
    # find out how many values the group variable can take
    levels = sorted(set(groupvariable))
    numlevels = len(levels)
    # get the mean and standard deviation for each group:
    numtotal = 0
```

```
    denomtotal = 0
    for leveli in levels:
        levelidata = variable[groupvariable==leveli]
        levelilength = len(levelidata)
        # get the standard deviation for group i:
        sdi = np.std(levelidata)
        numi = (levelilength)*sdi**2
        denomi = levelilength
        numtotal = numtotal + numi
        denomtotal = denomtotal + denomi
    # calculate the within-groups variance
    Vw = numtotal / (denomtotal - numlevels)
    return Vw
```

The `variable` parameter of the function `calcWithinGroupsVariance()` is the input variable for which we wish to compute its within-groups variance for the groups given in `groupvariable`.

So for example, to calculate the within-groups variance of the variable V2 (the concentration of the first chemical), we type:

```
calcWithinGroupsVariance(X.V2, y)
```

```
0.2620524691539065
```

Thus, the within-groups variance for V2 is 0.2620525.

We can calculate the between-groups variance for a particular variable (eg. V2) using the function `calcBetweenGroupsVariance()` below:

```
def calcBetweenGroupsVariance(variable, groupvariable):
    # find out how many values the group variable can take
    levels = sorted(set((groupvariable)))
    numlevels = len(levels)
    # calculate the overall grand mean:
    grandmean = np.mean(variable)
    # get the mean and standard deviation for each group:
    numtotal = 0
    denomtotal = 0
    for leveli in levels:
        levelidata = variable[groupvariable==leveli]
        levelilength = len(levelidata)
        # get the mean and standard deviation for group i:
        meani = np.mean(levelidata)
        sdi = np.std(levelidata)
        numi = levelilength * ((meani - grandmean)**2)
        denomi = levelilength
        numtotal = numtotal + numi
        denomtotal = denomtotal + denomi
    # calculate the between-groups variance
    Vb = numtotal / (numlevels - 1)
    return(Vb)
```

Similarly to the parameters of the function `calcWithinGroupsVariance()`, the `variable` parameter of the function `calcBetweenGroupsVariance()` is the input variable for which we wish to compute its between-groups variance for the groups given in `groupvariable`.

So for example, to calculate the between-groups variance of the variable V2 (the concentration of the first chemical), we type:

```
calcBetweenGroupsVariance(X.V2, y)
```

```
35.397424960269106
```

Thus, the between-groups variance of V2 is 35.397425.

We can calculate the *separation* achieved by a variable as its between-groups variance devided by its within-groups variance. Thus, the separation achieved by V2 is calculated as:

```
# 35.397424960269106 / 0.2620524691539065
calcBetweenGroupsVariance(X.V2, y) / calcWithinGroupsVariance(X.V2, y)
```

```
135.07762424279917
```

If you want to calculate the separations achieved by all of the variables in a multivariate data set, you can use the function `calcSeparations()` below:

```
def calcSeparations(variables, groupvariable):
    # calculate the separation for each variable
    for variablename in variables:
        variablei = variables[variablename]
        Vw = calcWithinGroupsVariance(variablei, groupvariable)
        Vb = calcBetweenGroupsVariance(variablei, groupvariable)
        sep = Vb/Vw
        print("variable", variablename, "Vw=", Vw, "Vb=", Vb, "separation=", sep)
```

For example, to calculate the separations for each of the 13 chemical concentrations, we type:

```
calcSeparations(X, y)
```

```
variable V2 Vw= 0.262052469154 Vb= 35.3974249603 separation= 135.077624243
variable V3 Vw= 0.887546796747 Vb= 32.7890184869 separation= 36.9434249632
variable V4 Vw= 0.0660721013425 Vb= 0.879611357249 separation= 13.3129012
variable V5 Vw= 8.00681118121 Vb= 286.416746363 separation= 35.7716374073
variable V6 Vw= 180.657773164 Vb= 2245.50102789 separation= 12.4295843381
variable V7 Vw= 0.191270475224 Vb= 17.9283572943 separation= 93.7330096204
variable V8 Vw= 0.274707514337 Vb= 64.2611950236 separation= 233.925872682
variable V9 Vw= 0.0119117022133 Vb= 0.328470157462 separation= 27.575417147
variable V10 Vw= 0.246172943796 Vb= 7.45199550778 separation= 30.2713831702
variable V11 Vw= 2.28492308133 Vb= 275.708000822 separation= 120.664018441
variable V12 Vw= 0.0244876469432 Vb= 2.48100991494 separation= 101.31679539
variable V13 Vw= 0.160778729561 Vb= 30.5435083544 separation= 189.972320579
variable V14 Vw= 29707.6818705 Vb= 6176832.32228 separation= 207.920373902
```

Thus, the individual variable which gives the greatest separations between the groups (the wine cultivars) is V8 (separation 233.9). As we will discuss below, the purpose of linear discriminant analysis (LDA) is to find the linear combination of the individual variables that will give the greatest separation between the groups (cultivars here). This hopefully will give a better separation than the best separation achievable by any individual variable (233.9 for V8 here).

Between-groups Covariance and Within-groups Covariance for Two Variables

If you have a multivariate data set with several variables describing sampling units from different groups, such as the wine samples from different cultivars, it is often of interest to calculate the within-groups covariance and between-groups variance for pairs of the variables.

This can be done using the following functions:

```
def calcWithinGroupsCovariance(variable1, variable2, groupvariable):
    levels = sorted(set(groupvariable))
    numlevels = len(levels)
    Covw = 0.0
    # get the covariance of variable 1 and variable 2 for each group:
    for leveli in levels:
        levelidata1 = variable1[groupvariable==leveli]
        levelidata2 = variable2[groupvariable==leveli]
        mean1 = np.mean(levelidata1)
        mean2 = np.mean(levelidata2)
        levelilength = len(levelidata1)
        # get the covariance for this group:
        term1 = 0.0
        for levelidata1j, levelidata2j in zip(levelidata1, levelidata2):
            term1 += (levelidata1j - mean1)*(levelidata2j - mean2)
        Cov_groupi = term1 # covariance for this group
        Covw += Cov_groupi
    totallength = len(variable1)
    Covw /= totallength - numlevels
    return Covw
```

For example, to calculate the within-groups covariance for variables V8 and V11, we type:

```
calcWithinGroupsCovariance(X.V8, X.V11, y)
```

```
0.28667830215140183
```

```
def calcBetweenGroupsCovariance(variable1, variable2, groupvariable):
    # find out how many values the group variable can take
    levels = sorted(set(groupvariable))
    numlevels = len(levels)
    # calculate the grand means
    variable1mean = np.mean(variable1)
    variable2mean = np.mean(variable2)
    # calculate the between-groups covariance
    Covb = 0.0
    for leveli in levels:
        levelidata1 = variable1[groupvariable==leveli]
        levelidata2 = variable2[groupvariable==leveli]
        mean1 = np.mean(levelidata1)
        mean2 = np.mean(levelidata2)
        levelilength = len(levelidata1)
        term1 = (mean1 - variable1mean) * (mean2 - variable2mean) * levelilength
        Covb += term1
    Covb /= numlevels - 1
    return Covb
```

For example, to calculate the between-groups covariance for variables V8 and V11, we type:

```
calcBetweenGroupsCovariance(X.V8, X.V11, y)
```

```
-60.4107748359163
```

Thus, for V8 and V11, the between-groups covariance is -60.41 and the within-groups covariance is 0.29. Since the within-groups covariance is positive (0.29), it means V8 and V11 are positively related within groups: for individuals from the same group, individuals with a high value of V8 tend to have a high value of V11, and vice versa. Since the between-groups covariance is negative (-60.41), V8 and V11 are negatively related between groups: groups with a high mean value of V8 tend to have a low mean value of V11, and vice versa.

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
```

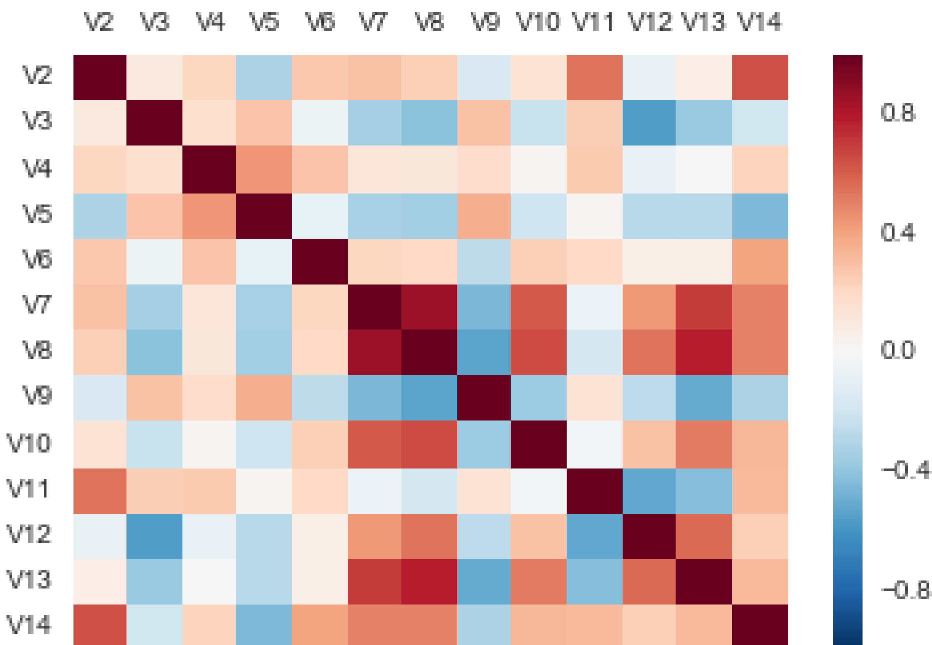
This tells us that the correlation coefficient is about 0.094, which is a very weak correlation. Furthermore, the *p-value* for the statistical test of whether the correlation coefficient is significantly different from zero is 0.21. This is much greater than 0.05 (which we can use here as a cutoff for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

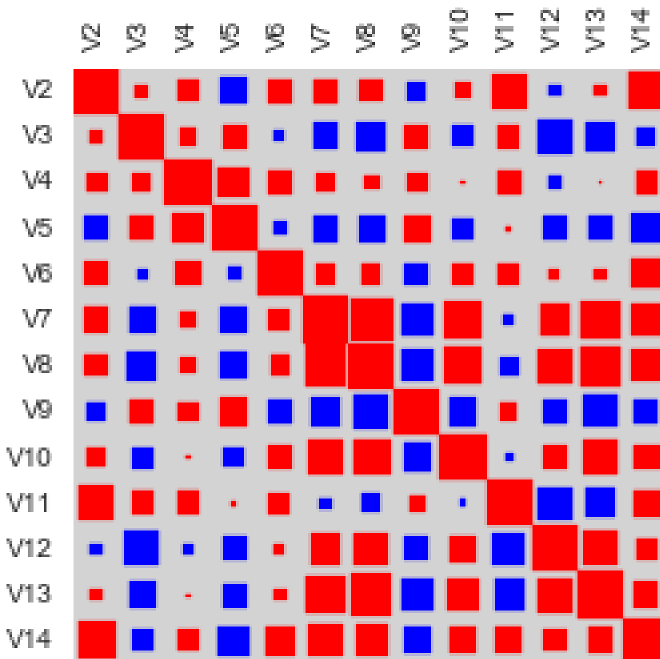
```
sns.heatmap(corrmat, vmax=1., square=False).xaxis.tick_top()
```



Or an alternative nice visualization is via a Hinton diagram. The color of the boxes determines the sign of the correlation, in this case red for positive and blue for negative correlations; while the size of the boxes determines their magnitude, the bigger the box the higher the magnitude.

```
# adapted from http://matplotlib.org/examples/specialty_plots/hinton_demo.html
def hinton(matrix, max_weight=None, ax=None):
```

```
    """Draw Hinton diagram for visualizing a weight matrix."""
    ax = ax if ax is not None else plt.gca()

    if not max_weight:
        max_weight = 2**np.ceil(np.log(np.abs(matrix).max())/np.log(2))

    ax.patch.set_facecolor('lightgray')
    ax.set_aspect('equal', 'box')
    ax.xaxis.set_major_locator(plt.NullLocator())
    ax.yaxis.set_major_locator(plt.NullLocator())

    for (x, y), w in np.ndenumerate(matrix):
        color = 'red' if w > 0 else 'blue'
        size = np.sqrt(np.abs(w))
        rect = plt.Rectangle([x - size / 2, y - size / 2], size, size,
                             facecolor=color, edgecolor=color)
        ax.add_patch(rect)

    nticks = matrix.shape[0]
    ax.xaxis.tick_top()
    ax.set_xticks(range(nticks))
    ax.set_xticklabels(list(matrix.columns), rotation=90)
    ax.set_yticks(range(nticks))
    ax.set_yticklabels(matrix.columns)
    ax.grid(False)

    ax.autoscale_view()
    ax.invert_yaxis()

hinton(corrmat)
```



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_index()
    # 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)
```

This tells us that the pair of variables with the highest linear correlation coefficient are V7 and V8 (correlation = 0.86 approximately).

Standardising Variables

If you want to compare different variables that have different units, are very different variances, it is a good idea to first standardise the variables.

For example, we found above that the concentrations of the 13 chemicals in the wine samples show a wide range of standard deviations, from 0.124103 for V9 (variance 0.015402) to 314.021657 for V14 (variance 98609.60). This is a range of approximately 6,402,389-fold in the variances.

As a result, it is not a good idea to use the unstandardised chemical concentrations as the input for a principal component analysis (PCA, see below) of the wine samples, as if you did that, the first principal component would be dominated by the variables which show the largest variances, such as V14.

Thus, it would be a better idea to first standardise the variables so that they all have variance 1 and mean 0, and to then carry out the principal component analysis on the standardised data. This would allow us to find the principal components that provide the best low-dimensional representation of the variation in the original data, without being overly biased by those variables that show the most variance in the original data.

You can standardise variables by using the `scale()` function from the package `sklearn.preprocessing`.

For example, to standardise the concentrations of the 13 chemicals in the wine samples, we type:

```
standardisedX = scale(X)
standardisedX = pd.DataFrame(standardisedX, index=X.index, columns=X.columns)
```

```
standardisedX.apply(np.mean)
```

```
V2    -8.619821e-16
V3    -8.357859e-17
V4    -8.657245e-16
```

```
V5    -1.160121e-16
V6    -1.995907e-17
V7    -2.972030e-16
V8    -4.016762e-16
V9     4.079134e-16
V10   -1.699639e-16
V11   -1.247442e-18
V12    3.717376e-16
V13    2.919013e-16
V14   -7.484650e-18
dtype: float64
```

```
standardisedX.apply(np.std)
```

```
V2      1
V3      1
V4      1
V5      1
V6      1
V7      1
V8      1
V9      1
V10     1
V11     1
V12     1
V13     1
V14     1
dtype: float64
```

2.1.5 Principal Component Analysis

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
    columns = pd.MultiIndex.from_tuples([("sdev", "Standard deviation"), ("varprop", "Proportion of V
    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)
```

```
Importance of components:
```

This gives us the standard deviation of each component, and the proportion of variance explained by each component. The standard deviation of the components is stored in a named row called `sdev` of the output variable made by the `pca_summary` function and stored in the `summary` variable:

```
summary.sdev
```

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

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

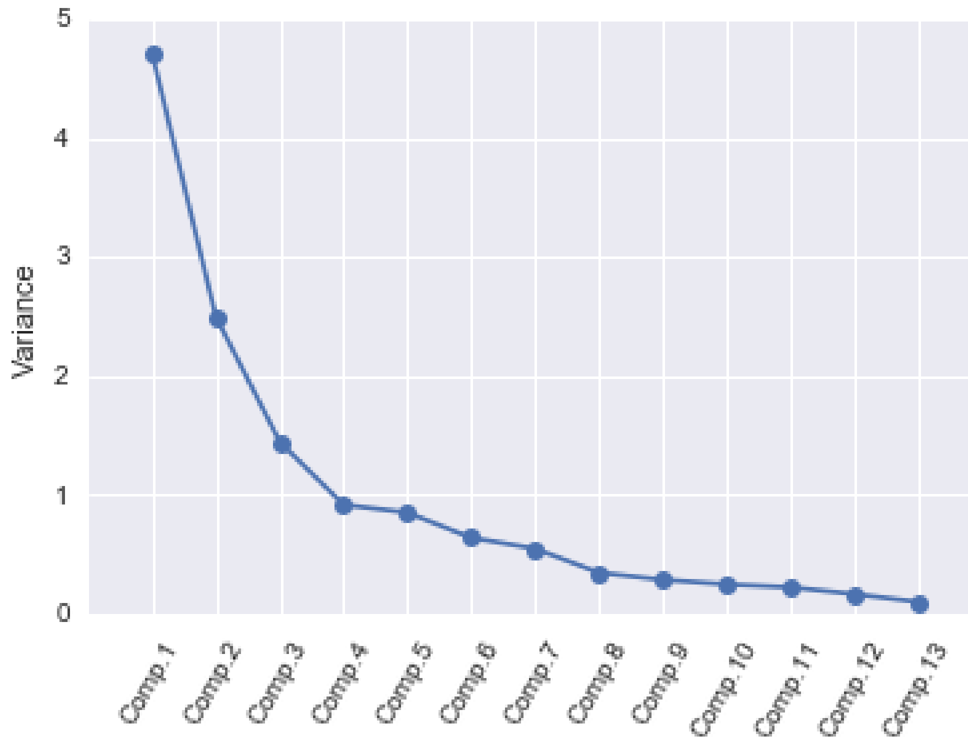
```
Standard deviation    13
dtype: float64
```

In 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. The total variance is equal to the sum of the variances of the individual variables, and since the variance of each standardised variable is 1, the total variance should be equal to the number of variables (13 here).

Deciding How Many Principal Components to Retain

In order to decide how many principal components should be retained, it is common to summarise the results of a principal components analysis by making a scree plot, which we can do using the `screeplot()` function below:

```
def screeplot(pca, standardised_values):
    y = np.std(pca.transform(standardised_values), axis=0)**2
    x = np.arange(len(y)) + 1
    plt.plot(x, y, "o-")
    plt.xticks(x, ["Comp."+str(i) for i in x], rotation=60)
    plt.ylabel("Variance")
    plt.show()

screeplot(pca, standardisedX)
```



The most obvious change in slope in the scree plot occurs at component 4, which is the "elbow" of the scree plot. Therefore, it cound be argued based on the basis of the scree plot that the first three components should be retained.

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.

A third way to decide how many principal components to retain is to decide to keep the number of components required to explain at least some minimum amount of the total variance. For example, if it is important to explain at least 80% of the variance, we would retain the first five principal components, as we can see from cumulative proportions (`summary.cumprop`) that the first five principal components explain 80.2% of the variance (while the first four components explain just 73.6%, so are not sufficient).

Loadings for the Principal Components

The loadings for the principal components are stored in a named element `components_` of the variable returned by `PCA().fit()`. This contains a matrix with the loadings of each principal component, where the first column in the matrix contains the loadings for the first principal component, the second column contains the loadings for the second principal component, and so on.

Therefore, to obtain the loadings for the first principal component in our analysis of the 13 chemical concentrations in wine samples, we type:

```
pca.components_[0]
```

```
array([-0.1443294 ,  0.24518758,  0.00205106,  0.23932041, -0.14199204,
       -0.39466085, -0.4229343 ,  0.2985331 , -0.31342949,  0.0886167 ,
       -0.29671456, -0.37616741, -0.28675223])
```

This means that the first principal component is a linear combination of the variables:

```
-0.144*Z2 + 0.245*Z3 + 0.002*Z4 + 0.239*Z5 - 0.142*Z6 - 0.395*Z7 - 0.423*Z8 + 0.299*Z9 -0.313*Z10 + 
```

where Z2, Z3, Z4, ..., Z14 are the standardised versions of the variables V2, V3, V4, ..., V14 (that each have mean of 0 and variance of 1).

Note that the square of the loadings sum to 1, as this is a constraint used in calculating the loadings:

```
np.sum(pca.components_[0]**2)
```

```
1.0000000000000004
```

To calculate the values of the first principal component, we can define our own function to calculate a principal component given the loadings and the input variables' values:

```
def calcpc(variables, loadings):
    # find the number of samples in the data set and the number of variables
    numsamples, numvariables = variables.shape
    # make a vector to store the component
    pc = np.zeros(numsamples)
    # calculate the value of the component for each sample
    for i in range(numsamples):
        valuei = 0
        for j in range(numvariables):
            valueij = variables.iloc[i, j]
            loadingj = loadings[j]
            valuei = valuei + (valueij * loadingj)
        pc[i] = valuei
    return pc
```

We can then use the function to calculate the values of the first principal component for each sample in our wine data:

```
calcpc(standardisedX, pca.components_[0])
```

```
array([-3.31675081, -2.20946492, -2.51674015, -3.75706561, -1.00890849,
       -3.05025392, -2.44908967, -2.05943687, -2.5108743 , -2.75362819,
       -3.47973668, -1.7547529 , -2.11346234, -3.45815682, -4.31278391,
       -2.3051882 , -2.17195527, -1.89897118, -3.54198508, -2.0845222 ,
       -3.12440254, -1.08657007, -2.53522408, -1.64498834, -1.76157587,
       -0.9900791 , -1.77527763, -1.23542396, -2.18840633, -2.25610898,
       -2.50022003, -2.67741105, -1.62857912, -1.90269086, -1.41038853,
       -1.90382623, -1.38486223, -1.12220741, -1.5021945 , -2.52980109,
       -2.58809543, -0.66848199, -3.07080699, -0.46220914, -2.10135193,
       -1.13616618, -2.72660096, -2.82133927, -2.00985085, -2.7074913 ,
       -3.21491747, -2.85895983, -3.50560436, -2.22479138, -2.14698782,
       -2.46932948, -2.74151791, -2.17374092, -3.13938015,  0.92858197,
        1.54248014,  1.83624976, -0.03060683, -2.05026161,  0.60968083,
       -0.90022784, -2.24850719, -0.18338403,  0.81280503, -1.9756205 ,
        1.57221622, -1.65768181,  0.72537239, -2.56222717, -1.83256757,
        0.8679929 , -0.3700144 ,  1.45737704, -1.26293085, -0.37615037,
       -0.7620639 , -1.03457797,  0.49487676,  2.53897708, -0.83532015,
       -0.78790461,  0.80683216,  0.55804262,  1.11511104,  0.55572283,
        1.34928528,  1.56448261,  1.93255561, -0.74666594, -0.95745536,
       -2.54386518,  0.54395259, -1.03104975, -2.25190942, -1.41021602,
       -0.79771979,  0.54953173,  0.16117374,  0.65979494, -0.39235441,
```

```
        1.77249908,  0.36626736,  1.62067257, -0.08253578, -1.57827507,
       -1.42056925,  0.27870275,  1.30314497,  0.45707187,  0.49418585,
       -0.48207441,  0.25288888,  0.10722764,  2.4330126 ,  0.55108954,
       -0.73962193, -1.33632173,  1.177087  ,  0.46233501, -0.97847408,
        0.09680973, -0.03848715,  1.5971585 ,  0.47956492,  1.79283347,
        1.32710166,  2.38450083,  2.9369401 ,  2.14681113,  2.36986949,
        3.06384157,  3.91575378,  3.93646339,  3.09427612,  2.37447163,
        2.77881295,  2.28656128,  2.98563349,  2.3751947 ,  2.20986553,
        2.625621  ,  4.28063878,  3.58264137,  2.80706372,  2.89965933,
        2.32073698,  2.54983095,  1.81254128,  2.76014464,  2.7371505 ,
        3.60486887,  2.889826  ,  3.39215608,  1.0481819 ,  1.60991228,
        3.14313097,  2.2401569 ,  2.84767378,  2.59749706,  2.94929937,
        3.53003227,  2.40611054,  2.92908473,  2.18141278,  2.38092779,
        3.21161722,  3.67791872,  2.4655558 ,  3.37052415,  2.60195585,
        2.67783946,  2.38701709,  3.20875816])
```

In fact, the values of the first principal component are computed with the following, so we can compare those values to the ones that we calculated, and they should agree:

```
pca.transform(standardisedX)[:, 0]
```

```
array([-3.31675081, -2.20946492, -2.51674015, -3.75706561, -1.00890849,
       -3.05025392, -2.44908967, -2.05943687, -2.5108743 , -2.75362819,
       -3.47973668, -1.7547529 , -2.11346234, -3.45815682, -4.31278391,
       -2.3051882 , -2.17195527, -1.89897118, -3.54198508, -2.0845222 ,
       -3.12440254, -1.08657007, -2.53522408, -1.64498834, -1.76157587,
       -0.9900791 , -1.77527763, -1.23542396, -2.18840633, -2.25610898,
       -2.50022003, -2.67741105, -1.62857912, -1.90269086, -1.41038853,
       -1.90382623, -1.38486223, -1.12220741, -1.5021945 , -2.52980109,
       -2.58809543, -0.66848199, -3.07080699, -0.46220914, -2.10135193,
       -1.13616618, -2.72660096, -2.82133927, -2.00985085, -2.7074913 ,
       -3.21491747, -2.85895983, -3.50560436, -2.22479138, -2.14698782,
       -2.46932948, -2.74151791, -2.17374092, -3.13938015,  0.92858197,
        1.54248014,  1.83624976, -0.03060683, -2.05026161,  0.60968083,
       -0.90022784, -2.24850719, -0.18338403,  0.81280503, -1.9756205 ,
        1.57221622, -1.65768181,  0.72537239, -2.56222717, -1.83256757,
        0.8679929 , -0.3700144 ,  1.45737704, -1.26293085, -0.37615037,
       -0.7620639 , -1.03457797,  0.49487676,  2.53897708, -0.83532015,
       -0.78790461,  0.80683216,  0.55804262,  1.11511104,  0.55572283,
        1.34928528,  1.56448261,  1.93255561, -0.74666594, -0.95745536,
       -2.54386518,  0.54395259, -1.03104975, -2.25190942, -1.41021602,
       -0.79771979,  0.54953173,  0.16117374,  0.65979494, -0.39235441,
        1.77249908,  0.36626736,  1.62067257, -0.08253578, -1.57827507,
       -1.42056925,  0.27870275,  1.30314497,  0.45707187,  0.49418585,
       -0.48207441,  0.25288888,  0.10722764,  2.4330126 ,  0.55108954,
       -0.73962193, -1.33632173,  1.177087  ,  0.46233501, -0.97847408,
        0.09680973, -0.03848715,  1.5971585 ,  0.47956492,  1.79283347,
        1.32710166,  2.38450083,  2.9369401 ,  2.14681113,  2.36986949,
        3.06384157,  3.91575378,  3.93646339,  3.09427612,  2.37447163,
        2.77881295,  2.28656128,  2.98563349,  2.3751947 ,  2.20986553,
        2.625621  ,  4.28063878,  3.58264137,  2.80706372,  2.89965933,
        2.32073698,  2.54983095,  1.81254128,  2.76014464,  2.7371505 ,
        3.60486887,  2.889826  ,  3.39215608,  1.0481819 ,  1.60991228,
        3.14313097,  2.2401569 ,  2.84767378,  2.59749706,  2.94929937,
        3.53003227,  2.40611054,  2.92908473,  2.18141278,  2.38092779,
        3.21161722,  3.67791872,  2.4655558 ,  3.37052415,  2.60195585,
        2.67783946,  2.38701709,  3.20875816])
```

We see that they do agree.

The first principal component has highest (in absolute value) loadings for V8 (-0.423), V7 (-0.395), V13 (-0.376), V10 (-0.313), V12 (-0.297), V14 (-0.287), V9 (0.299), V3 (0.245), and V5 (0.239). The loadings for V8, V7, V13, V10, V12 and V14 are negative, while those for V9, V3, and V5 are positive. Therefore, an interpretation of the first principal component is that it represents a contrast between the concentrations of V8, V7, V13, V10, V12, and V14, and the concentrations of V9, V3 and V5.

Similarly, we can obtain the loadings for the second principal component by typing:

```
pca.components_[1]
```

```
array([ 0.48365155,  0.22493093,  0.31606881, -0.0105905 ,  0.299634  ,
        0.06503951, -0.00335981,  0.02877949,  0.03930172,  0.52999567,
       -0.27923515, -0.16449619,  0.36490283])
```

This means that the second principal component is a linear combination of the variables:

```
0.484*Z2 + 0.225*Z3 + 0.316*Z4 - 0.011*Z5 + 0.300*Z6 + 0.065*Z7 - 0.003*Z8 + 0.029*Z9 + 0.039*Z10 +
```

where Z1, Z2, Z3, ..., Z14 are the standardised versions of variables V2, V3, ..., V14 that each have mean 0 and variance 1.

Note that the square of the loadings sum to 1, as above:

```
np.sum(pca.components_[1]**2)
```

```
1.0000000000000011
```

The second principal component has highest loadings for V11 (0.530), V2 (0.484), V14 (0.365), V4 (0.316), V6 (0.300), V12 (-0.279), and V3 (0.225). The loadings for V11, V2, V14, V4, V6 and V3 are positive, while the loading for V12 is negative. Therefore, an interpretation of the second principal component is that it represents a contrast between the concentrations of V11, V2, V14, V4, V6 and V3, and the concentration of V12. Note that the loadings for V11 (0.530) and V2 (0.484) are the largest, so the contrast is mainly between the concentrations of V11 and V2, and the concentration of V12.

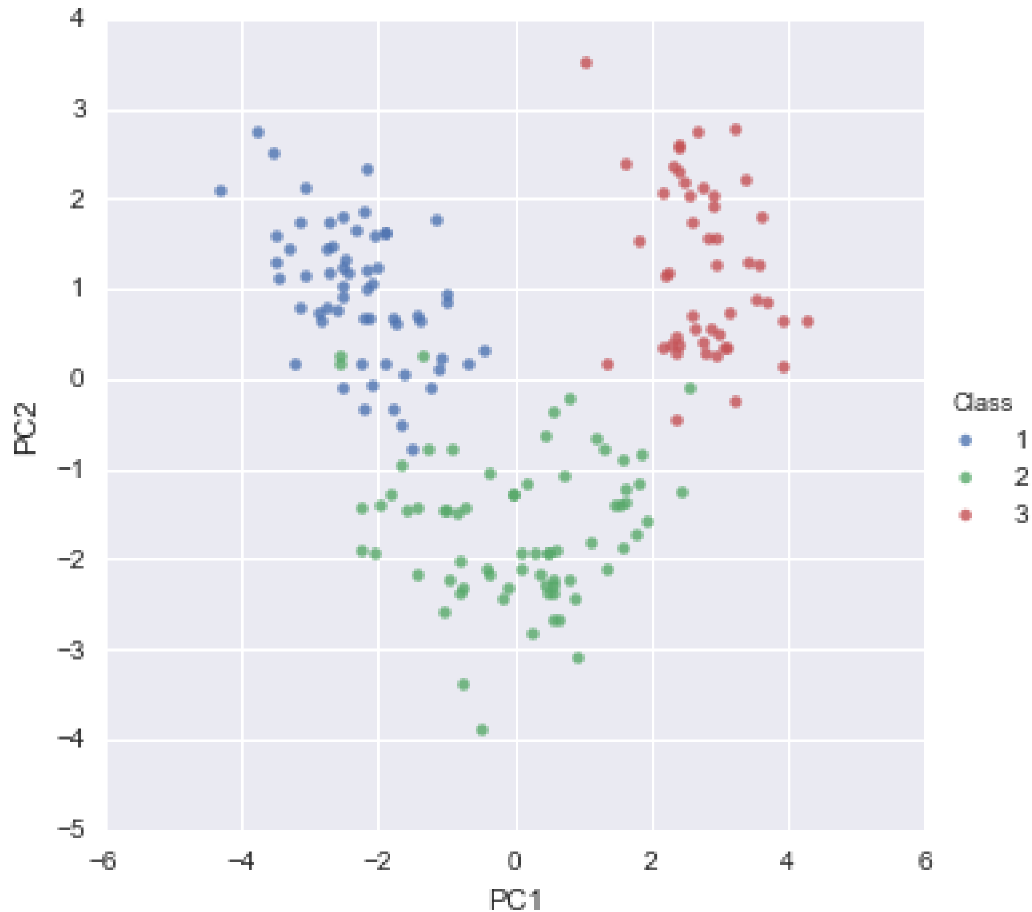
Scatterplots of the Principal Components

The values of the principal components can be computed by the `transform()` (or `fit_transform()`) method of the `PCA` class. It returns a matrix with the principal components, where the first column in the matrix contains the first principal component, the second column the second component, and so on.

Thus, in our example, `pca.transform(standardisedX)[:, 0]` contains the first principal component, and `pca.transform(standardisedX)[:, 1]` contains the second principal component.

We can make a scatterplot of the first two principal components, and label the data points with the cultivar that the wine samples come from, by typing:

```
def pca_scatter(pca, standardised_values, classifs):
    foo = pca.transform(standardised_values)
    bar = pd.DataFrame(zip(foo[:, 0], foo[:, 1], classifs), columns=["PC1", "PC2", "Class"])
    sns.lmplot("PC1", "PC2", bar, hue="Class", fit_reg=False)

pca_scatter(pca, standardisedX, y)
```



The scatterplot shows the first principal component on the x-axis, and the second principal component on the y-axis. We can see from the scatterplot that wine samples of cultivar 1 have much lower values of the first principal component than wine samples of cultivar 3. Therefore, the first principal component separates wine samples of cultivars 1 from those of cultivar 3.

We can also see that wine samples of cultivar 2 have much higher values of the second principal component than wine samples of cultivars 1 and 3. Therefore, the second principal component separates samples of cultivar 2 from samples of cultivars 1 and 3.

Therefore, the first two principal components are reasonably useful for distinguishing wine samples of the three different cultivars.

Above, we interpreted the first principal component as a contrast between the concentrations of V8, V7, V13, V10, V12, and V14, and the concentrations of V9, V3 and V5. We can check whether this makes sense in terms of the concentrations of these chemicals in the different cultivars, by printing out the means of the standardised concentration variables in each cultivar, using the `printMeanAndSdByGroup()` function (see above):

```
printMeanAndSdByGroup(standardisedX, y);
```

```
## Means:
```

```
## Standard deviations:
```

```
## Sample sizes:
```

Does it make sense that the first principal component can separate cultivar 1 from cultivar 3? In cultivar 1, the mean values of V8 (0.954), V7 (0.871), V13 (0.769), V10 (0.539), V12 (0.458) and V14 (1.171) are very high compared

to the mean values of V9 (-0.577), V3 (-0.292) and V5 (-0.736). In cultivar 3, the mean values of V8 (-1.249), V7 (-0.985), V13 (-1.307), V10 (-0.764), V12 (-1.202) and V14 (-0.372) are very low compared to the mean values of V9 (0.688), V3 (0.893) and V5 (0.575). Therefore, it does make sense that principal component 1 is a contrast between the concentrations of V8, V7, V13, V10, V12, and V14, and the concentrations of V9, V3 and V5; and that principal component 1 can separate cultivar 1 from cultivar 3.

Above, we intepreted the second principal component as a contrast between the concentrations of V11, V2, V14, V4, V6 and V3, and the concentration of V12. In the light of the mean values of these variables in the different cultivars, does it make sense that the second principal component can separate cultivar 2 from cultivars 1 and 3? In cultivar 1, the mean values of V11 (0.203), V2 (0.917), V14 (1.171), V4 (0.325), V6 (0.462) and V3 (-0.292) are not very different from the mean value of V12 (0.458). In cultivar 3, the mean values of V11 (1.009), V2 (0.189), V14 (-0.372), V4 (0.257), V6 (-0.030) and V3 (0.893) are also not very different from the mean value of V12 (-1.202). In contrast, in cultivar 2, the mean values of V11 (-0.850), V2 (-0.889), V14 (-0.722), V4 (-0.444), V6 (-0.364) and V3 (-0.361) are much less than the mean value of V12 (0.432). Therefore, it makes sense that principal component is a contrast between the concentrations of V11, V2, V14, V4, V6 and V3, and the concentration of V12; and that principal component 2 can separate cultivar 2 from cultivars 1 and 3.

2.1.6 Linear Discriminant Analysis

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. By carrying out a principal component analysis, we found that most of the variation in the chemical concentrations between the samples can be captured using the first two principal components, where each of the principal components is a particular linear combination of the 13 chemical concentrations.

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.sca
    if out:
        print("Coefficients of linear discriminants:")
        display(ret)
    return ret
```

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

```
Coefficients of linear discriminants:
```

This means that the first discriminant function is a linear combination of the variables:

```
-0.403*V2 + 0.165*V3 - 0.369*V4 + 0.155*V5 - 0.002*V6 + 0.618*V7 - 1.661*V8 - 1.496*V9 + 0.134*V10 +
```

where V2, V3, ..., V14 are the concentrations of the 14 chemicals found in the wine samples. For convenience, the value for each discriminant function (eg. the first discriminant function) are scaled so that their mean value is zero (see below).

Note that these loadings are calculated so that the within-group variance of each discriminant function for each group (cultivar) is equal to 1, as will be demonstrated below.

As mentioned above, these scalings are stored in the named member `scalings_` of the object variable returned by `LinearDiscriminantAnalysis().fit(X, y)`. This element contains a numpy array, in which the first column contains the loadings for the first discriminant function, the second column contains the loadings for the second discriminant function and so on. For example, to extract the loadings for the first discriminant function, we can type:

```
lda.scalings_[:, 0]
```

```
array([-0.40339978,  0.1652546 , -0.36907526,  0.15479789, -0.0021635 ,
        0.61805207, -1.66119123, -1.49581844,  0.13409263,  0.35505571,
       -0.81803607, -1.15755938, -0.00269121])
```

Or for "prettier" print, use the dataframe variable created above:

```
pretty_scalings_.LD1
```

```
V2    -0.403400
V3     0.165255
V4    -0.369075
V5     0.154798
V6    -0.002163
V7     0.618052
V8    -1.661191
V9    -1.495818
V10    0.134093
V11    0.355056
V12   -0.818036
V13   -1.157559
V14   -0.002691
Name: LD1, dtype: float64
```

To calculate the values of the first discriminant function, we can define our own function `calclda()`:

```
def calclda(variables, loadings):
    # find the number of samples in the data set and the number of variables
    numsamples, numvariables = variables.shape
    # make a vector to store the discriminant function
    ld = np.zeros(numsamples)
    # calculate the value of the discriminant function for each sample
    for i in range(numsamples):
        valuei = 0
        for j in range(numvariables):
            valueij = variables.iloc[i, j]
            loadingj = loadings[j]
            valuei = valuei + (valueij * loadingj)
```

```
        ld[i] = valuei
    # standardise the discriminant function so that its mean value is 0:
    ld = scale(ld, with_std=False)
    return ld
```

The function `calclda()` simply calculates the value of a discriminant function for each sample in the data set, for example, for the first disriminant function, for each sample we calculate the value using the equation:

```
-0.403*V2 - 0.165*V3 - 0.369*V4 + 0.155*V5 - 0.002*V6 + 0.618*V7 - 1.661*V8 - 1.496*V9 + 0.134*V10 +
```

Furthermore, the `scale()` command is used within the `calclda()` function in order to standardise the value of a discriminant function (eg. the first discriminant function) so that its mean value (over all the wine samples) is 0.

We can use the function `calclda()` to calculate the values of the first discriminant function for each sample in our wine data:

```
calclda(X, lda.scalings_[:, 0])
```

```
array([-4.70024401, -4.30195811, -3.42071952, -4.20575366, -1.50998168,
       -4.51868934, -4.52737794, -4.14834781, -3.86082876, -3.36662444,
       -4.80587907, -3.42807646, -3.66610246, -5.58824635, -5.50131449,
       -3.18475189, -3.28936988, -2.99809262, -5.24640372, -3.13653106,
       -3.57747791, -1.69077135, -4.83515033, -3.09588961, -3.32164716,
       -2.14482223, -3.9824285 , -2.68591432, -3.56309464, -3.17301573,
       -2.99626797, -3.56866244, -3.38506383, -3.5275375 , -2.85190852,
       -2.79411996, -2.75808511, -2.17734477, -3.02926382, -3.27105228,
       -2.92065533, -2.23721062, -4.69972568, -1.23036133, -2.58203904,
       -2.58312049, -3.88887889, -3.44975356, -2.34223331, -3.52062596,
       -3.21840912, -4.38214896, -4.36311727, -3.51917293, -3.12277475,
       -1.8024054 , -2.87378754, -3.61690518, -3.73868551,  1.58618749,
        0.79967216,  2.38015446, -0.45917726, -0.50726885,  0.39398359,
       -0.92256616, -1.95549377, -0.34732815,  0.20371212, -0.24831914,
        1.17987999, -1.07718925,  0.64100179, -1.74684421, -0.34721117,
        1.14274222,  0.18665882,  0.900525  , -0.70709551, -0.59562833,
       -0.55761818, -1.80430417,  0.23077079,  2.03482711, -0.62113021,
       -1.03372742,  0.76598781,  0.35042568,  0.15324508, -0.14962842,
        0.48079504,  1.39689016,  0.91972331, -0.59102937,  0.49411386,
       -1.62614426,  2.00044562, -1.00534818, -2.07121314, -1.6381589 ,
       -1.0589434 ,  0.02594549, -0.21887407,  1.3643764 , -1.12901245,
       -0.21263094, -0.77946884,  0.61546732,  0.22550192, -2.03869851,
        0.79274716,  0.30229545, -0.50664882,  0.99837397, -0.21954922,
       -0.37131517,  0.05545894, -0.09137874,  1.79755252, -0.17405009,
       -1.17870281, -3.2105439 ,  0.62605202,  0.03366613, -0.6993008 ,
       -0.72061079, -0.51933512,  1.17030045,  0.10824791,  1.12319783,
        2.24632419,  3.28527755,  4.07236441,  3.86691235,  3.45088333,
        3.71583899,  3.9222051 ,  4.8516102 ,  3.54993389,  3.76889174,
        2.6694225 ,  2.32491492,  3.17712883,  2.88964418,  3.78325562,
        3.04411324,  4.70697017,  4.85021393,  4.98359184,  4.86968293,
        4.5986919 ,  5.67447884,  5.32986123,  5.03401031,  4.52080087,
        5.0978371 ,  5.04368277,  4.86980829,  5.61316558,  5.67046737,
        5.37413513,  3.09975377,  3.35888137,  3.04007194,  4.94861303,
        4.54504458,  5.27255844,  5.13016117,  4.30468082,  5.08336782,
        4.06743571,  5.74212961,  4.4820514 ,  4.29150758,  4.50329623,
        5.04747033,  4.27615505,  5.5380861 ])
```

In fact, the values of the first linear discriminant function can be calculated using the `transform(X)` or `fit_transform(X, y)` methods of the LDA object, so we can compare those to the ones that we calculated, and they should agree:

```
# Try either, they produce the same result, use help() for more info
# lda.transform(X)[:, 0]
lda.fit_transform(X, y)[:, 0]
```

```
array([-4.70024401, -4.30195811, -3.42071952, -4.20575366, -1.50998168,
       -4.51868934, -4.52737794, -4.14834781, -3.86082876, -3.36662444,
       -4.80587907, -3.42807646, -3.66610246, -5.58824635, -5.50131449,
       -3.18475189, -3.28936988, -2.99809262, -5.24640372, -3.13653106,
       -3.57747791, -1.69077135, -4.83515033, -3.09588961, -3.32164716,
       -2.14482223, -3.9824285 , -2.68591432, -3.56309464, -3.17301573,
       -2.99626797, -3.56866244, -3.38506383, -3.5275375 , -2.85190852,
       -2.79411996, -2.75808511, -2.17734477, -3.02926382, -3.27105228,
       -2.92065533, -2.23721062, -4.69972568, -1.23036133, -2.58203904,
       -2.58312049, -3.88887889, -3.44975356, -2.34223331, -3.52062596,
       -3.21840912, -4.38214896, -4.36311727, -3.51917293, -3.12277475,
       -1.8024054 , -2.87378754, -3.61690518, -3.73868551,  1.58618749,
        0.79967216,  2.38015446, -0.45917726, -0.50726885,  0.39398359,
       -0.92256616, -1.95549377, -0.34732815,  0.20371212, -0.24831914,
        1.17987999, -1.07718925,  0.64100179, -1.74684421, -0.34721117,
        1.14274222,  0.18665882,  0.900525  , -0.70709551, -0.59562833,
       -0.55761818, -1.80430417,  0.23077079,  2.03482711, -0.62113021,
       -1.03372742,  0.76598781,  0.35042568,  0.15324508, -0.14962842,
        0.48079504,  1.39689016,  0.91972331, -0.59102937,  0.49411386,
       -1.62614426,  2.00044562, -1.00534818, -2.07121314, -1.6381589 ,
       -1.0589434 ,  0.02594549, -0.21887407,  1.3643764 , -1.12901245,
       -0.21263094, -0.77946884,  0.61546732,  0.22550192, -2.03869851,
        0.79274716,  0.30229545, -0.50664882,  0.99837397, -0.21954922,
       -0.37131517,  0.05545894, -0.09137874,  1.79755252, -0.17405009,
       -1.17870281, -3.2105439 ,  0.62605202,  0.03366613, -0.6993008 ,
       -0.72061079, -0.51933512,  1.17030045,  0.10824791,  1.12319783,
        2.24632419,  3.28527755,  4.07236441,  3.86691235,  3.45088333,
        3.71583899,  3.9222051 ,  4.8516102 ,  3.54993389,  3.76889174,
        2.6694225 ,  2.32491492,  3.17712883,  2.88964418,  3.78325562,
        3.04411324,  4.70697017,  4.85021393,  4.98359184,  4.86968293,
        4.5986919 ,  5.67447884,  5.32986123,  5.03401031,  4.52080087,
        5.0978371 ,  5.04368277,  4.86980829,  5.61316558,  5.67046737,
        5.37413513,  3.09975377,  3.35888137,  3.04007194,  4.94861303,
        4.54504458,  5.27255844,  5.13016117,  4.30468082,  5.08336782,
        4.06743571,  5.74212961,  4.4820514 ,  4.29150758,  4.50329623,
        5.04747033,  4.27615505,  5.5380861 ])
```

We see that they do agree.

It doesn't matter whether the input variables for linear discriminant analysis are standardised or not, unlike for principal components analysis in which it is often necessary to standardise the input variables. However, using standardised variables in linear discriminant analysis makes it easier to interpret the loadings in a linear discriminant function.

In linear discriminant analysis, the standardised version of an input variable is defined so that it has mean zero and within-groups variance of 1. Thus, we can calculate the "group-standardised" variable by subtracting the mean from each value of the variable, and dividing by the within-groups standard deviation. To calculate the group-standardised version of a set of variables, we can use the function `groupStandardise()` below:

```
def groupStandardise(variables, groupvariable):
    # find the number of samples in the data set and the number of variables
    numsamples, numvariables = variables.shape
    # find the variable names
    variablenames = variables.columns
    # calculate the group-standardised version of each variable
    variables_new = pd.DataFrame()
```

```
    for i in range(numvariables):
        variable_name = variablenames[i]
        variablei = variables[variable_name]
        variablei_Vw = calcWithinGroupsVariance(variablei, groupvariable)
        variablei_mean = np.mean(variablei)
        variablei_new = (variablei - variablei_mean)/(np.sqrt(variablei_Vw))
        variables_new[variable_name] = variablei_new
    return variables_new
```

For example, we can use the `groupStandardise()` function to calculate the group-standardised versions of the chemical concentrations in wine samples:

```
groupstandardisedX = groupStandardise(X, y)
```

We can then use the `LinearDiscriminantAnalysis().fit()` method to perform linear disriminant analysis on the group-standardised variables:

```
lda2 = LinearDiscriminantAnalysis().fit(groupstandardisedX, y)
pretty_scalings(lda2, groupstandardisedX)
```

It makes sense to interpret the loadings calculated using the group-standardised variables rather than the loadings for the original (unstandardised) variables.

In the first discriminant function calculated for the group-standardised variables, the largest loadings (in absolute) value are given to V8 (-0.871), V11 (0.537), V13 (-0.464), V14 (-0.464), and V5 (0.438). The loadings for V8, V13 and V14 are negative, while those for V11 and V5 are positive. Therefore, the discriminant function seems to represent a contrast between the concentrations of V8, V13 and V14, and the concentrations of V11 and V5.

We saw above that the individual variables which gave the greatest separations between the groups were V8 (separation 233.93), V14 (207.92), V13 (189.97), V2 (135.08) and V11 (120.66). These were mostly the same variables that had the largest loadings in the linear discriminant function (loading for V8: -0.871, for V14: -0.464, for V13: -0.464, for V11: 0.537).

We found above that variables V8 and V11 have a negative between-groups covariance (-60.41) and a positive within-groups covariance (0.29). When the between-groups covariance and within-groups covariance for two variables have opposite signs, it indicates that a better separation between groups can be obtained by using a linear combination of those two variables than by using either variable on its own.

Thus, given that the two variables V8 and V11 have between-groups and within-groups covariances of opposite signs, and that these are two of the variables that gave the greatest separations between groups when used individually, it is not surprising that these are the two variables that have the largest loadings in the first discriminant function.

Note that although the loadings for the group-standardised variables are easier to interpret than the loadings for the unstandardised variables, the values of the discriminant function are the same regardless of whether we standardise the input variables or not. For example, for wine data, we can calculate the value of the first discriminant function calculated using the unstandardised and group-standardised variables by typing:

```
lda.fit_transform(X, y)[:, 0]
```

```
array([-4.70024401, -4.30195811, -3.42071952, -4.20575366, -1.50998168,
       -4.51868934, -4.52737794, -4.14834781, -3.86082876, -3.36662444,
       -4.80587907, -3.42807646, -3.66610246, -5.58824635, -5.50131449,
       -3.18475189, -3.28936988, -2.99809262, -5.24640372, -3.13653106,
       -3.57747791, -1.69077135, -4.83515033, -3.09588961, -3.32164716,
       -2.14482223, -3.9824285 , -2.68591432, -3.56309464, -3.17301573,
       -2.99626797, -3.56866244, -3.38506383, -3.5275375 , -2.85190852,
       -2.79411996, -2.75808511, -2.17734477, -3.02926382, -3.27105228,
       -2.92065533, -2.23721062, -4.69972568, -1.23036133, -2.58203904,
       -2.58312049, -3.88887889, -3.44975356, -2.34223331, -3.52062596,
```

```
       -3.21840912, -4.38214896, -4.36311727, -3.51917293, -3.12277475,
       -1.8024054 , -2.87378754, -3.61690518, -3.73868551,  1.58618749,
        0.79967216,  2.38015446, -0.45917726, -0.50726885,  0.39398359,
       -0.92256616, -1.95549377, -0.34732815,  0.20371212, -0.24831914,
        1.17987999, -1.07718925,  0.64100179, -1.74684421, -0.34721117,
        1.14274222,  0.18665882,  0.900525  , -0.70709551, -0.59562833,
       -0.55761818, -1.80430417,  0.23077079,  2.03482711, -0.62113021,
       -1.03372742,  0.76598781,  0.35042568,  0.15324508, -0.14962842,
        0.48079504,  1.39689016,  0.91972331, -0.59102937,  0.49411386,
       -1.62614426,  2.00044562, -1.00534818, -2.07121314, -1.6381589 ,
       -1.0589434 ,  0.02594549, -0.21887407,  1.3643764 , -1.12901245,
       -0.21263094, -0.77946884,  0.61546732,  0.22550192, -2.03869851,
        0.79274716,  0.30229545, -0.50664882,  0.99837397, -0.21954922,
       -0.37131517,  0.05545894, -0.09137874,  1.79755252, -0.17405009,
       -1.17870281, -3.2105439 ,  0.62605202,  0.03366613, -0.6993008 ,
       -0.72061079, -0.51933512,  1.17030045,  0.10824791,  1.12319783,
        2.24632419,  3.28527755,  4.07236441,  3.86691235,  3.45088333,
        3.71583899,  3.9222051 ,  4.8516102 ,  3.54993389,  3.76889174,
        2.6694225 ,  2.32491492,  3.17712883,  2.88964418,  3.78325562,
        3.04411324,  4.70697017,  4.85021393,  4.98359184,  4.86968293,
        4.5986919 ,  5.67447884,  5.32986123,  5.03401031,  4.52080087,
        5.0978371 ,  5.04368277,  4.86980829,  5.61316558,  5.67046737,
        5.37413513,  3.09975377,  3.35888137,  3.04007194,  4.94861303,
        4.54504458,  5.27255844,  5.13016117,  4.30468082,  5.08336782,
        4.06743571,  5.74212961,  4.4820514 ,  4.29150758,  4.50329623,
        5.04747033,  4.27615505,  5.5380861 ])
```

```
lda2.fit_transform(groupstandardisedX, y)[:, 0]
```

```
array([-4.70024401, -4.30195811, -3.42071952, -4.20575366, -1.50998168,
       -4.51868934, -4.52737794, -4.14834781, -3.86082876, -3.36662444,
       -4.80587907, -3.42807646, -3.66610246, -5.58824635, -5.50131449,
       -3.18475189, -3.28936988, -2.99809262, -5.24640372, -3.13653106,
       -3.57747791, -1.69077135, -4.83515033, -3.09588961, -3.32164716,
       -2.14482223, -3.9824285 , -2.68591432, -3.56309464, -3.17301573,
       -2.99626797, -3.56866244, -3.38506383, -3.5275375 , -2.85190852,
       -2.79411996, -2.75808511, -2.17734477, -3.02926382, -3.27105228,
       -2.92065533, -2.23721062, -4.69972568, -1.23036133, -2.58203904,
       -2.58312049, -3.88887889, -3.44975356, -2.34223331, -3.52062596,
       -3.21840912, -4.38214896, -4.36311727, -3.51917293, -3.12277475,
       -1.8024054 , -2.87378754, -3.61690518, -3.73868551,  1.58618749,
        0.79967216,  2.38015446, -0.45917726, -0.50726885,  0.39398359,
       -0.92256616, -1.95549377, -0.34732815,  0.20371212, -0.24831914,
        1.17987999, -1.07718925,  0.64100179, -1.74684421, -0.34721117,
        1.14274222,  0.18665882,  0.900525  , -0.70709551, -0.59562833,
       -0.55761818, -1.80430417,  0.23077079,  2.03482711, -0.62113021,
       -1.03372742,  0.76598781,  0.35042568,  0.15324508, -0.14962842,
        0.48079504,  1.39689016,  0.91972331, -0.59102937,  0.49411386,
       -1.62614426,  2.00044562, -1.00534818, -2.07121314, -1.6381589 ,
       -1.0589434 ,  0.02594549, -0.21887407,  1.3643764 , -1.12901245,
       -0.21263094, -0.77946884,  0.61546732,  0.22550192, -2.03869851,
        0.79274716,  0.30229545, -0.50664882,  0.99837397, -0.21954922,
       -0.37131517,  0.05545894, -0.09137874,  1.79755252, -0.17405009,
       -1.17870281, -3.2105439 ,  0.62605202,  0.03366613, -0.6993008 ,
       -0.72061079, -0.51933512,  1.17030045,  0.10824791,  1.12319783,
        2.24632419,  3.28527755,  4.07236441,  3.86691235,  3.45088333,
        3.71583899,  3.9222051 ,  4.8516102 ,  3.54993389,  3.76889174,
        2.6694225 ,  2.32491492,  3.17712883,  2.88964418,  3.78325562,
```

```
        3.04411324,  4.70697017,  4.85021393,  4.98359184,  4.86968293,
        4.5986919 ,  5.67447884,  5.32986123,  5.03401031,  4.52080087,
        5.0978371 ,  5.04368277,  4.86980829,  5.61316558,  5.67046737,
        5.37413513,  3.09975377,  3.35888137,  3.04007194,  4.94861303,
        4.54504458,  5.27255844,  5.13016117,  4.30468082,  5.08336782,
        4.06743571,  5.74212961,  4.4820514 ,  4.29150758,  4.50329623,
        5.04747033,  4.27615505,  5.5380861 ])
```

We can see that although the loadings are different for the first discriminant functions calculated using unstandardised and group-standardised data, the actual values of the first discriminant function are the same.

Separation Achieved by the Discriminant Functions

To calculate the separation achieved by each discriminant function, we first need to calculate the value of each discriminant function, by substituting the values of the variables into the linear combination for the discriminant function (eg. `-0.403*V2 - 0.165*V3 - 0.369*V4 + 0.155*V5 - 0.002*V6 + 0.618*V7 - 1.661*V8 - 1.496*V9 + 0.134*V10 + 0.355*V11 - 0.818*V12 - 1.158*V13 - 0.003*V14` for the first discriminant function), and then scaling the values of the discriminant function so that their mean is zero.

As mentioned above, we can do this using the `rpredict()` function which simulates the output of the `predict()` function in R. For example, to calculate the value of the discriminant functions for the wine data, we type:

```
def rpredict(lda, X, y, out=False):
    ret = {"class": lda.predict(X),
           "posterior": pd.DataFrame(lda.predict_proba(X), columns=lda.classes_)}
    ret["x"] = pd.DataFrame(lda.fit_transform(X, y))
    ret["x"].columns = ["LD"+str(i+1) for i in range(ret["x"].shape[1])]
    if out:
        print("class")
        print(ret["class"])
        print()
        print("posterior")
        print(ret["posterior"])
        print()
        print("x")
        print(ret["x"])
    return ret

lda_values = rpredict(lda, standardisedX, y, True)
```

```
class
['2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2'
 '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2'
 '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2'
 '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2'
 '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2'
 '2' '2' '2' '2' '2' '2' '3' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2'
 '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2'
 '2' '2' '2' '2' '3' '3' '3' '3' '2' '2' '2' '2' '2' '2' '2' '2' '2' '2'
 '3' '2' '3' '3' '3' '3' '3' '3' '3' '2' '2' '2' '3' '2' '3' '3' '2' '2'
 '2' '2' '3' '2' '3' '3' '3' '3' '2' '2' '3' '3' '3' '3' '2' '3']

posterior
                1         2         3
0    1.344367e-22  0.999236  0.000764
1    4.489007e-27  0.983392  0.016608
2    2.228888e-24  0.791616  0.208384
```

```
3    1.026755e-24  0.500161  0.499839
4    6.371860e-23  0.790657  0.209343
5    1.552082e-24  0.981986  0.018014
6    3.354960e-23  0.951823  0.048177
7    3.417899e-22  0.925154  0.074846
8    4.041139e-26  0.978998  0.021002
9    3.718868e-26  0.619841  0.380159
..            ...       ...       ...
168  7.463695e-30  0.500000  0.500000
169  1.389203e-29  0.499927  0.500073
170  1.356187e-33  0.500000  0.500000
171  1.007615e-33  0.500000  0.500000
172  1.524219e-30  0.500000  0.500000
173  1.317492e-30  0.500000  0.500000
174  2.664128e-32  0.500000  0.500000
175  2.873436e-34  0.500000  0.500000
176  1.479166e-32  0.500000  0.500000
177  1.209888e-28  0.500000  0.500000

[178 rows x 3 columns]

x
          LD1       LD2
0   -4.700244  1.979138
1   -4.301958  1.170413
2   -3.420720  1.429101
3   -4.205754  4.002871
4   -1.509982  0.451224
5   -4.518689  3.213138
6   -4.527378  3.269122
7   -4.148348  3.104118
8   -3.860829  1.953383
9   -3.366624  1.678643
..        ...       ...
168  4.304681  2.391125
169  5.083368  3.157667
170  4.067436  0.318922
171  5.742130  1.467082
172  4.482051  3.307084
173  4.291508  3.390332
174  4.503296  2.083546
175  5.047470  3.196231
176  4.276155  2.431388
177  5.538086  3.042057

[178 rows x 2 columns]
```

The returned variable has a named element `x` which is a matrix containing the linear discriminant functions: the first column of `x` contains the first discriminant function, the second column of `x` contains the second discriminant function, and so on (if there are more discriminant functions).

We can therefore calculate the separations achieved by the two linear discriminant functions for the wine data by using the `calcSeparations()` function (see above), which calculates the separation as the ratio of the between-groups variance to the within-groups variance:

```
calcSeparations(lda_values["x"], y)
```

```
variable LD1 Vw= 1.0 Vb= 794.652200566 separation= 794.652200566
variable LD2 Vw= 1.0 Vb= 361.241041493 separation= 361.241041493
```

As mentioned above, the loadings for each discriminant function are calculated in such a way that the within-group variance (`Vw`) for each group (wine cultivar here) is equal to 1, as we see in the output from `calcSeparations()` above.

The output from `calcSeparations()` tells us that the separation achieved by the first (best) discriminant function is 794.7, and the separation achieved by the second (second best) discriminant function is 361.2.

Therefore, the total separation is the sum of these, which is (`794.652200566216+361.241041493455=1155.893`) 1155.89, rounded to two decimal places. Therefore, the *percentage separation* achieved by the first discriminant function is (`794.652200566216*100/1155.893=`) 68.75%, and the percentage separation achieved by the second discriminant function is (`361.241041493455*100/1155.893=`) 31.25%.

The *proportion of trace* (as reported in R by the `lda()` model) is the percentage separation achieved by each discriminant function. For example, for the wine data we get the same values as just calculated (68.75% and 31.25%). Note that in `sklearn` the proportion of trace is reported as `explained_variance_ratio_` in a `LinearDiscriminantAnalysis` model and is computed only for an "eigen" solver, while so far we have been using the default one, which is "svd" (Singular Value Decomposition):

```
def proportion_of_trace(lda):
    ret = pd.DataFrame([round(i, 4) for i in lda.explained_variance_ratio_ if round(i, 4) > 0], colur
    ret.index = ["LD"+str(i+1) for i in range(ret.shape[0])]
    ret = ret.transpose()
    print("Proportion of trace:")
    print(ret.to_string(index=False))
    return ret

proportion_of_trace(LinearDiscriminantAnalysis(solver="eigen").fit(X, y));
```

```
Proportion of trace:
    LD1     LD2
 0.6875  0.3125
```

Therefore, the first discriminant function does achieve a good separation between the three groups (three cultivars), but the second discriminant function does improve the separation of the groups by quite a large amount, so is it worth using the second discriminant function as well. Therefore, to achieve a good separation of the groups (cultivars), it is necessary to use both of the first two discriminant functions.

We found above that the largest separation achieved for any of the individual variables (individual chemical concentrations) was 233.9 for V8, which is quite a lot less than 794.7, the separation achieved by the first discriminant function. Therefore, the effect of using more than one variable to calculate the discriminant function is that we can find a discriminant function that achieves a far greater separation between groups than achieved by any one variable alone.

A Stacked Histogram of the LDA Values

A nice way of displaying the results of a linear discriminant analysis (LDA) is to make a stacked histogram of the values of the discriminant function for the samples from different groups (different wine cultivars in our example).

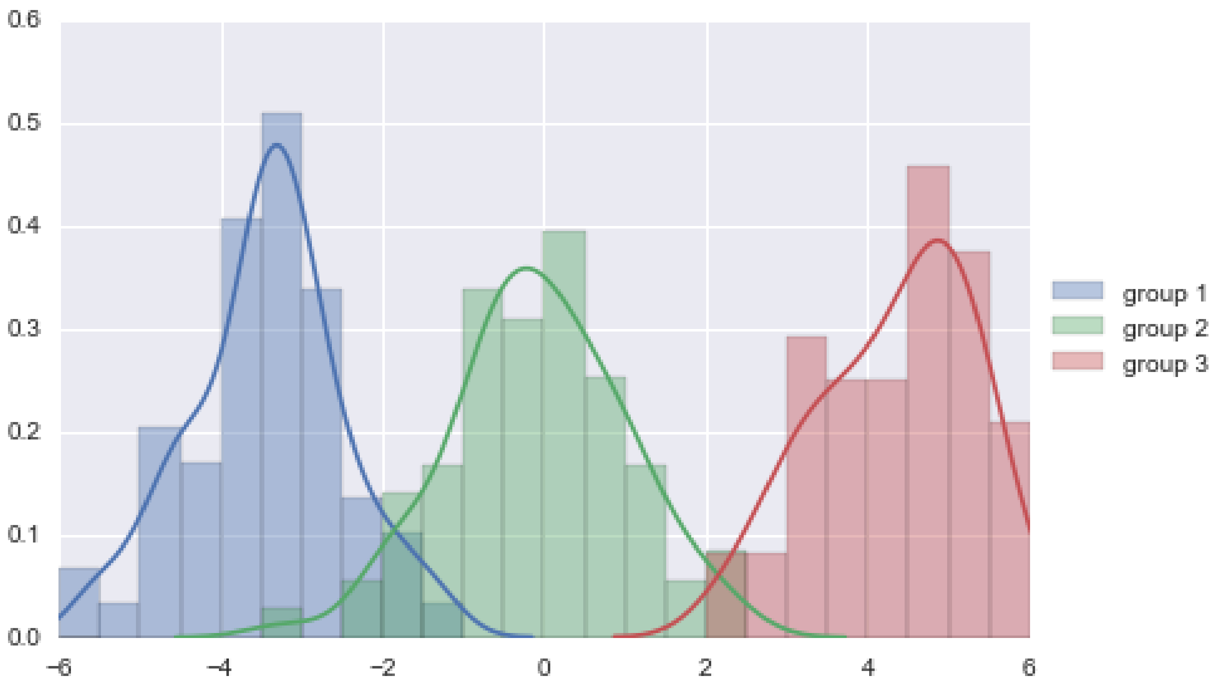
We can do this using the `ldahist()` function defined below.

```
def ldahist(data, g, sep=False):
    xmin = np.trunc(np.min(data)) - 1
    xmax = np.trunc(np.max(data)) + 1
    ncol = len(set(g))
    binwidth = 0.5
```

```
    bins=np.arange(xmin, xmax + binwidth, binwidth)
    if sep:
        fig, axl = plt.subplots(ncol, 1, sharey=True, sharex=True)
    else:
        fig, axl = plt.subplots(1, 1, sharey=True, sharex=True)
        axl = [axl]*ncol
    for ax, (group, gdata) in zip(axl, data.groupby(g)):
        sns.distplot(gdata.values, bins, ax=ax, label="group "+str(group))
        ax.set_xlim([xmin, xmax])
        if sep:
            ax.set_xlabel("group"+str(group))
        else:
            ax.legend(loc='center left', bbox_to_anchor=(1, 0.5))
    plt.tight_layout()
```

For example, to make a stacked histogram of the first discriminant function's values for wine samples of the three different wine cultivars, we type:

```
ldahist(lda_values["x"].LD1, y)
```



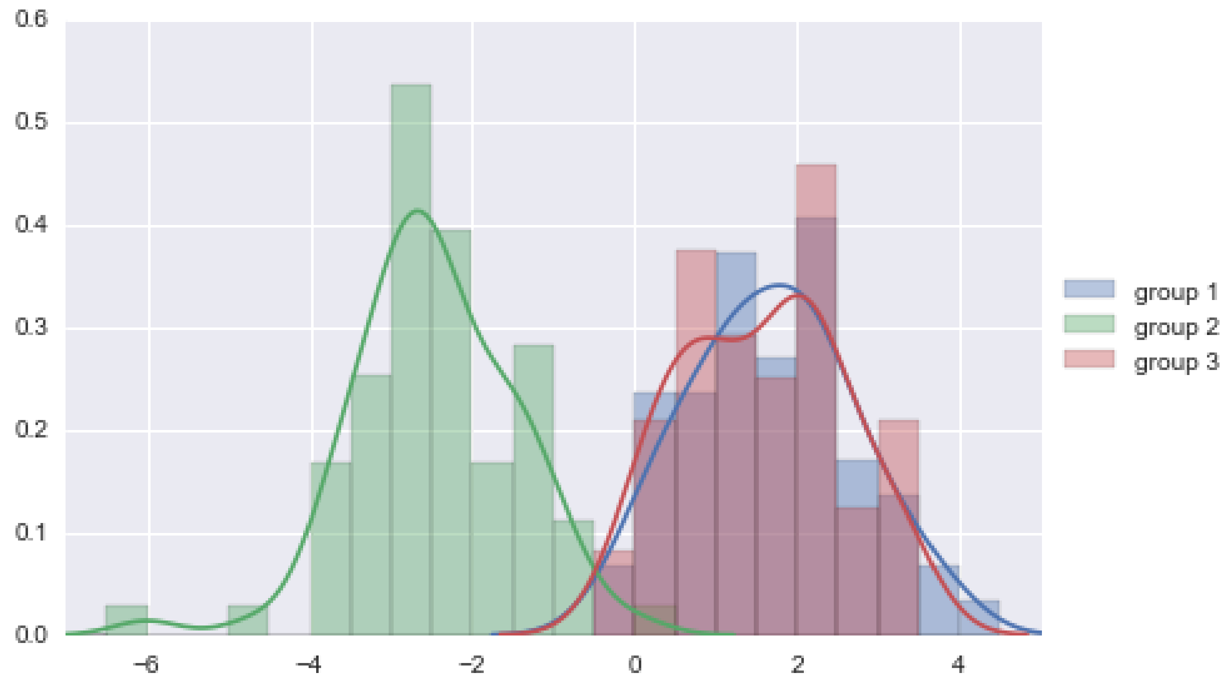
We can see from the histogram that cultivars 1 and 3 are well separated by the first discriminant function, since the values for the first cultivar are between -6 and -1, while the values for cultivar 3 are between 2 and 6, and so there is no overlap in values.

However, the separation achieved by the linear discriminant function on the training set may be an overestimate. To get a more accurate idea of how well the first discriminant function separates the groups, we would need to see a stacked histogram of the values for the three cultivars using some unseen "test set", that is, using a set of data that was not used to calculate the linear discriminant function.

We see that the first discriminant function separates cultivars 1 and 3 very well, but does not separate cultivars 1 and 2, or cultivars 2 and 3, so well.

We therefore investigate whether the second discriminant function separates those cultivars, by making a stacked histogram of the second discriminant function's values:

```
ldahist(lda_values["x"].LD2, y)
```



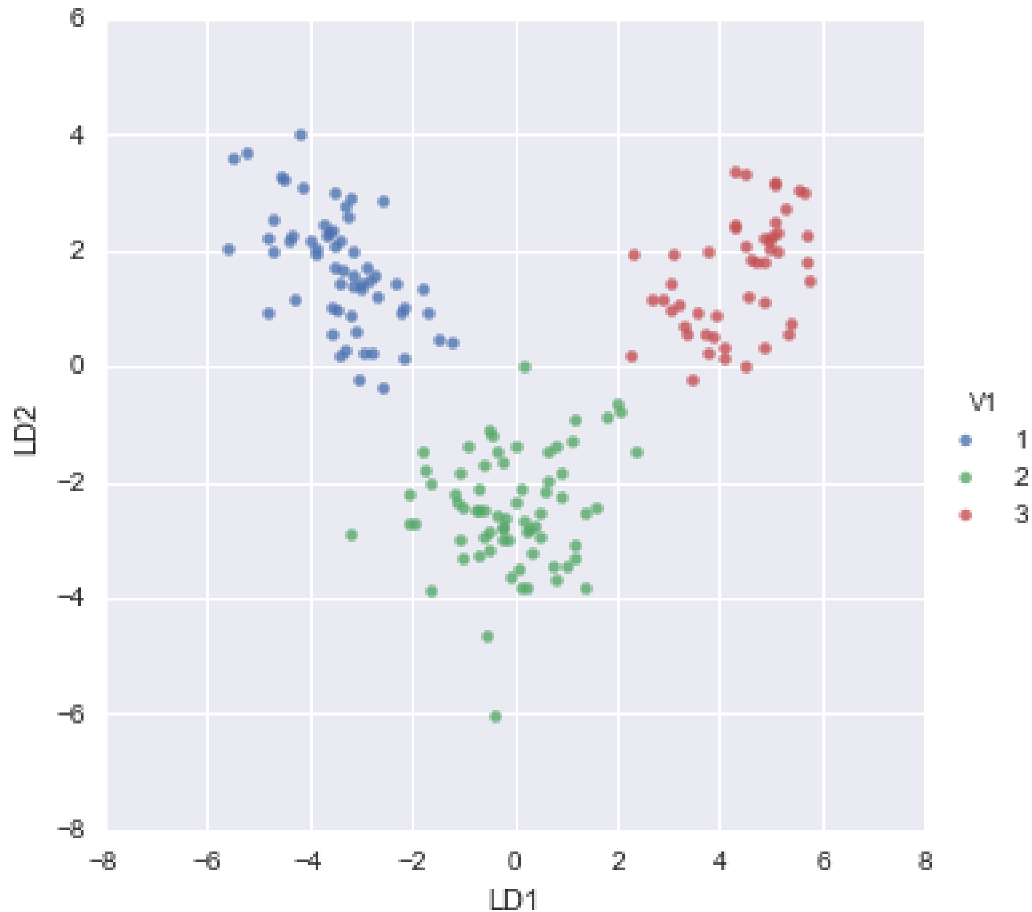
We see that the second discriminant function separates cultivars 1 and 2 quite well, although there is a little overlap in their values. Furthermore, the second discriminant function also separates cultivars 2 and 3 quite well, although again there is a little overlap in their values so it is not perfect.

Thus, we see that two discriminant functions are necessary to separate the cultivars, as was discussed above (see the discussion of percentage separation above).

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);
```



From the scatterplot of the first two discriminant functions, we can see that the wines from the three cultivars are well separated in the scatterplot. The first discriminant function (x-axis) separates cultivars 1 and 3 very well, but doesn't not perfectly separate cultivars 1 and 3, or cultivars 2 and 3.

The second discriminant function (y-axis) achieves a fairly good separation of cultivars 1 and 3, and cultivars 2 and 3, although it is not totally perfect.

To achieve a very good separation of the three cultivars, it would be best to use both the first and second discriminant functions together, since the first discriminant function can separate cultivars 1 and 3 very well, and the second discriminant function can separate cultivars 1 and 2, and cultivars 2 and 3, reasonably well.

Allocation Rules and Misclassification Rate

We can calculate the mean values of the discriminant functions for each of the three cultivars using the `printMeanAndSdByGroup()` function (see above):

```
printMeanAndSdByGroup(lda_values["x"], y);
```

```
## Means:
```

```
## Standard deviations:
```

```
## Sample sizes:
```

We find that the mean value of the first discriminant function is -3.42248851 for cultivar 1, -0.07972623 for cultivar 2, and 4.32473717 for cultivar 3. The mid-way point between the mean values for cultivars 1 and 2 is (-3.42248851-0.07972623)/2=-1.751107, and the mid-way point between the mean values for cultivars 2 and 3 is (-0.07972623+4.32473717)/2 = 2.122505.

Therefore, we can use the following allocation rule:

- if the first discriminant function is <= -1.751107, predict the sample to be from cultivar 1
- if the first discriminant function is > -1.751107 and <= 2.122505, predict the sample to be from cultivar 2
- if the first discriminant function is > 2.122505, predict the sample to be from cultivar 3

We can examine the accuracy of this allocation rule by using the `calcAllocationRuleAccuracy()` function below:

```
def calcAllocationRuleAccuracy(ldavalue, groupvariable, cutoffpoints):
    # find out how many values the group variable can take
    levels = sorted(set((groupvariable)))
    numlevels = len(levels)
    confusion_matrix = []
    # calculate the number of true positives and false negatives for each group
    for i, leveli in enumerate(levels):
        levelidata = ldavalue[groupvariable==leveli]
        row = []
        # see how many of the samples from this group are classified in each group
        for j, levelj in enumerate(levels):
            if j == 0:
                cutoff1 = cutoffpoints[0]
                cutoff2 = "NA"
                results = (levelidata <= cutoff1).value_counts()
            elif j == numlevels-1:
                cutoff1 = cutoffpoints[numlevels-2]
                cutoff2 = "NA"
                results = (levelidata > cutoff1).value_counts()
            else:
                cutoff1 = cutoffpoints[j-1]
                cutoff2 = cutoffpoints[j]
                results = ((levelidata > cutoff1) & (levelidata <= cutoff2)).value_counts()
            try:
                trues = results[True]
            except KeyError:
                trues = 0
            print("Number of samples of group", leveli, "classified as group", levelj, ":", trues, "
            row.append(trues)
        confusion_matrix.append(row)
    return confusion_matrix
```

For example, to calculate the accuracy for the wine data based on the allocation rule for the first discriminant function, we type:

```
confusion_matrix = calcAllocationRuleAccuracy(lda_values["x"].iloc[:, 0], y, [-1.751107, 2.122505])
```

```
Number of samples of group 1 classified as group 1 : 56 (cutoffs: -1.751107 , NA )
Number of samples of group 1 classified as group 2 : 3 (cutoffs: -1.751107 , 2.122505 )
Number of samples of group 1 classified as group 3 : 0 (cutoffs: 2.122505 , NA )
Number of samples of group 2 classified as group 1 : 5 (cutoffs: -1.751107 , NA )
Number of samples of group 2 classified as group 2 : 65 (cutoffs: -1.751107 , 2.122505 )
Number of samples of group 2 classified as group 3 : 1 (cutoffs: 2.122505 , NA )
Number of samples of group 3 classified as group 1 : 0 (cutoffs: -1.751107 , NA )
```

```
Number of samples of group 3 classified as group 2 : 0 (cutoffs: -1.751107 , 2.122505 )
Number of samples of group 3 classified as group 3 : 48 (cutoffs: 2.122505 , NA )
```

This can be displayed in a *confusion matrix*:

```
def webprint_confusion_matrix(confusion_matrix, classes_names):
    display(pd.DataFrame(confusion_matrix, index=["Is group "+i for i in classes_names], columns=["Al

webprint_confusion_matrix(confusion_matrix, lda.classes_)
```

There are 3+5+1=9 wine samples that are misclassified, out of (56+3+5+65+1+48=) 178 wine samples: 3 samples from cultivar 1 are predicted to be from cultivar 2, 5 samples from cultivar 2 are predicted to be from cultivar 1, and 1 sample from cultivar 2 is predicted to be from cultivar 3. Therefore, the misclassification rate is 9/178, or 5.1%. The misclassification rate is quite low, and therefore the accuracy of the allocation rule appears to be relatively high.

However, this is probably an underestimate of the misclassification rate, as the allocation rule was based on this data (this is the *training set*). If we calculated the misclassification rate for a separate *test set* consisting of data other than that used to make the allocation rule, we would probably get a higher estimate of the misclassification rate.

The Python way

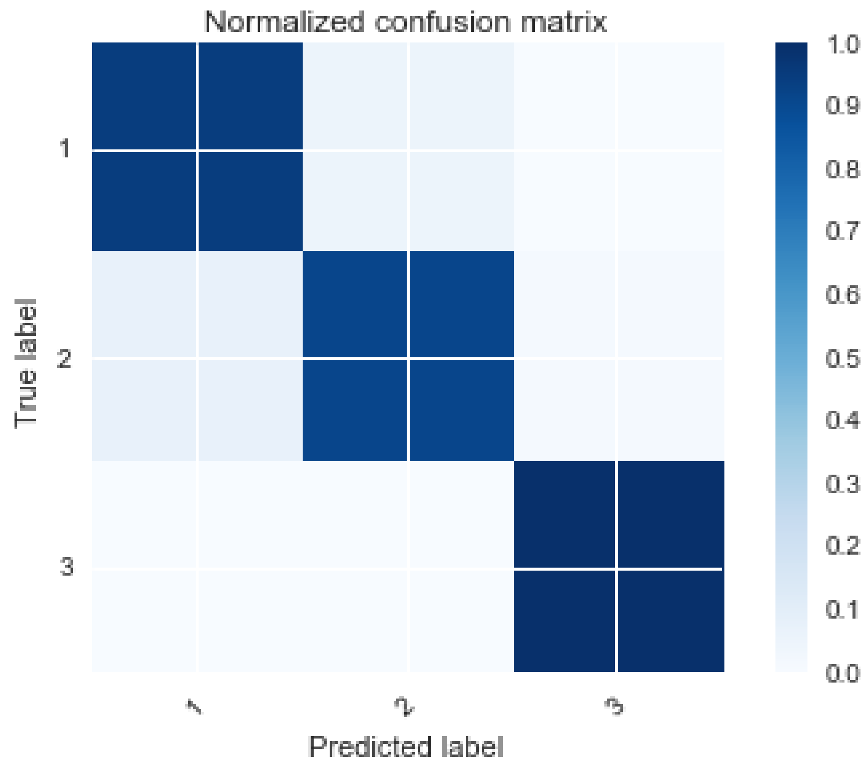
Python allows to do all the above in a much faster way and providing extended automatic report capabilities by using the `sklearn.metrics` module. The above confusion matrix and reporting typical performance metrics, such as *precision*, *recall*, *f1-score* can be done in python as follows:

```
import sklearn.metrics as metrics

def lda_classify(v, levels, cutoffpoints):
    for level, cutoff in zip(reversed(levels), reversed(cutoffpoints)):
        if v > cutoff: return level
    return levels[0]

y_pred = lda_values["x"].iloc[:, 0].apply(lda_classify, args=(lda.classes_, [-1.751107, 2.122505],))
y_true = y
```

```
# from http://scikit-learn.org/stable/auto_examples/model_selection/plot_confusion_matrix.html#examp
def plot_confusion_matrix(cm, target_names, title='Confusion matrix', cmap=plt.cm.Blues):
    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title)
    plt.colorbar()
    tick_marks = np.arange(len(target_names))
    plt.xticks(tick_marks, target_names, rotation=45)
    plt.yticks(tick_marks, target_names)
    plt.tight_layout()
    plt.ylabel('True label')
    plt.xlabel('Predicted label')

print(metrics.classification_report(y_true, y_pred))
cm = metrics.confusion_matrix(y_true, y_pred)
webprint_confusion_matrix(cm, lda.classes_)
cm_normalized = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
plot_confusion_matrix(cm_normalized, lda.classes_, title='Normalized confusion matrix')
```

```
             precision    recall  f1-score   support

          1       0.92      0.95      0.93        59
          2       0.96      0.92      0.94        71
```

```
          3       0.98      1.00      0.99        48

avg / total       0.95      0.95      0.95       178
```



2.1.7 Links and Further Reading

Here are some info and links for further reading.

To learn about multivariate analysis I would recommend the following:

- Multivariate Data Analysis by Hair et. al.
- Applied Multivariate Data Analysis by Everitt and Dunn.

If you are new to python then you can read one of the overwhelming number of tutorials that exist on the web. Here are a couple of links:

- The official tutorial by the Python organization. Extensive covering pretty much everything in core python and with lots of detailed non-interactive examples.
- For those that prefer learning by doing through interactive tutorials here are some good ones I would recommend:
  - Codeacademy Python tutorial
  - learnpython.org

To learn about data analysis and data science using the Python ecosystem I would recommend the following:

- Python for Data Analysis by Wes McKinney
- Data Science from Scratch by Joel Grus

To learn about doing machine learning in Python using scikit-learn I would recommend:

- The scikit-learn homepage not only has excellent documentation and examples, but also provides useful and clear resources about machine learning methods.
- Python Machine Learning by Sebastian Raschka

The visualizations here were produced using matplotlib and seaborn. Their homepages have extensive documentation about their APIs and tons of examples.

Although Python is self-sufficient, fairly extensive and probably growing more rapidly than any other scientific language, if the need arises that you need to use R for whatever reason, then you are able to do so within Python itself via the `rpy2` library.

2.1.8 Acknowledgements

I would like to thank Avril Coghlan, Wellcome Trust Sanger Institute, Cambridge, U.K. for her excellent resource A Little Book of R for Multivariate Analysis and releasing it under a CC-BY-3.0 License, hence, allowing this translation from R to Python. All kudos to her.

As the original, many of the examples in this booklet are inspired by examples in the Open University book, "Multivariate Analysis" (product code M249/03).

I am also grateful to the UCI Machine Learning Repository, http://archive.ics.uci.edu/ml, for making data sets available which were used in the examples in this booklet.

2.1.9 Contact

I will be grateful if you will send me (Yiannis Gatsoulis) corrections or suggestions for improvements to my email address gatsoulis AT gmail DOT com.

2.1.10 License

A Little Book of Python for Multivariate Analysis by Yiannis Gatsoulis is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.Based on a work at A Little Book of R for Multivariate Analysis by Avril Coghlan licensed under CC-BY-3.0.

CHAPTER 3

License