Handling files

Machine Learning: Lab Sheet 1

Miguel Juarez <m.juarez@sheffield.ac.uk>

Data formatting

Data can be stored in a lot of formats, and the data scientist may need to deal with many. Data often needs "cleaning", that is, we need to deal with missing observations, incomplete data, data entered in the wrong format etc. Cleaning data is a big part of the data scientist's job, but we shall say little about it, and usually take our data to be in a convenient structured format.

Let's say a little more about what this might mean. "Structured format" might involve some sort of tabular data, where every row represents an observation, and every column a variable. This is the sort of thing you would have in an Excel spreadsheet. Data in this form is known (in the R community, at least) as "tidy". So for example, we might have the data set

	revision	sleep	mark
Module 1	9	8	67
Module 2	12	7	79
Module 3	8	6	71

as in Chapter 1 of the notes, representing the mark on three modules, compared with the number of hours of revision and number of hours of sleep.

You can imagine this data stored as an Excel spreadsheet. We could store it as a plain text file, as above, but care needs to be taken about spaces and tabs.

Comma separated value files

A better way to store the information in a plain text file is as a comma separated value (CSV, or `.csv`) file, where we separate entries by commas:

```
,revision,sleep,mark
Module 1,9,8,67
Module 2,12,7,79
Module 3,8,6,71
```

You may be wondering what happens if your entries have commas in. We can enclose entries in quotation marks – for example if "Module 1" might be "Data Science, Machine Learning and Artificial Intelligence", you will need to enclose the module name in quotation marks `"..."`. Alternatively, you may prefer to edit your data so that there are no commas, but this is often fiddly. You may also prefer a different *separator*; an alternative is tab-separated files, where tabs are used as separators, but commas are most commonly used.

We will tend to work with CSV files, since both can easily be imported into Python and to R. But you may need to convert your data set into this form. Other ways to store data include HTML/XML files (so web-based), or JSON (again, web-based, using JavaScript Object Notation), or as a relational database, using SQL, or a non-relational database ("NoSQL"). There are ways to convert between these formats; some are easy to do – you can save an Excel file as a `.csv` file easily.

Dealing with tidying data isn't the main focus of this module, but if you do want to ask me more, feel free. For converting files, I find the pattern matching language AWK (see https://www.gnu.org/software/gawk/ and especially the Windows 64-bit download at https://www.klabaster.

com/freeware.htm) to be very useful. The authors of the language, Aho, Weinberger and Kernighan, have a nice book on it. But some plain text editors like `emacs`, `notepad` and `vim` also have some capabilities. And `.csv` files naturally open in Excel, and one can do a lot of data manipulation with Excel's commands. Finally, several of R's `tidyverse` packages are useful for tidying data (see the book *R for Data Science* by Grolemund and Wickham, for example), and there are almost certainly Python alternatives; possibly it is more natural to do everything either within R or within Python.

For the rest of the module, we'll work only with datasets given in CSV format, so that they can be opened by anyone. Furthermore, these will all be in "tidy" format, and will have no missing values. (In real life, the data scientist will need to deal with these issues!)

The file `airpoll.csv`

For this lab sheet, we will be working with the CSV file `airpoll.csv` from the Blackboard page for the course. The data set originates with Brian Everitt's book, "An R and S-plus Companion to Multivariate Analysis", and is the main dataset considered in the second chapter of that book (Everitt attributes the data to McDonald and Schwing, 1973). Here are the first few lines of the file:

```
"","Rainfall","Education","Popden","Nonwhite","NOX","SO2","Mortality"
"akronOH",36,11.4,3243,8.8,15,59,921.9
"albanyNY",35,11,4281,3.5,10,39,997.9
"allenPA",44,9.8,4260,0.8,6,33,962.4
```

It contains data on 60 cities in the USA, starting with Akron, Ohio, and Albany, New York, and contains variables:

- `Rainfall` mean annual precipitation in inches
- `Education` median school years completed for those over 25 in 1960
- `Popden` population per square mile in urbanised area in 1960
- `Nonwhite` percentage of urban area population that is nonwhite
- `NOX` relative pollution potential of oxides of nitrogen
- `SO2` relative pollution potential of sulphur dioxide
- `Mortality` total age-adjusted mortality rate, expressed as deaths per 100,000

We'll use this data set for this lab sheet.

1 Lab Sheet 1a: Handling Files in Python

1.1 The Jupyter notebook

The Jupyter notebook is a standard way in data science to run Python. First, in the Windows start menu, find the directory "Anaconda 3", and go to "Jupyter notebook", if it exists (if you installed `miniconda`, you may need to install the `jupyter` package). After a while, an interactive window will open in your web browser. (There are other ways to do this, in Spyder, IDLE, etc., but Jupyter notebooks are fairly common in the data science and machine learning communities.) The window which opens is a directory listing – find out which one it is; it is the *working directory* (see below). Mine is stored in `C:\Users\pm1afj` by default, but this may work differently on the University managed desktop.

Near the top right of the window is a box labelled `New` – click on that: if you installed a virtual environment as suggested in Lab Sheet 0, you should see an option "Python (ML)" in the `Notebook` option here, which you should select. (Otherwise, simply select "Python 3" from the `Notebook` option.) You will get a new notebook. Type `1+1` into the obvious box, and click on the `Run` box on the line above. This input is then processed by Python, which returns `2` as the output. You should get the idea that commands are run in these cells, with the output printed by the notebook. (You can add text in the "Markdown" language by typing `m` at the left of the cell, where it has "`In [ ]`", before clicking on `Run`: LaTeX is allowed as part of Markdown as well.)

Let's save this – using the `File` menu, click on "Save and Checkpoint", and then click again on `File` and "Rename" it as, say, `ML_test`. In the directory where Jupyter notebooks are stored, you should now see a file `ML_test.ipynb`.

You could email this to yourself, and on another computer, save it in the appropriate place, open up "Jupyter notebook" as above, and can then open `ML_test.ipynb` to see that the whole notebook has all been copied across.

The Jupyter notebook has a working directory; you can put files in the working directory, and access them simply by their file names. My working directory is `C:\Users\pm1afj`, but I can access files in other places by including the full path (see below).

Remark. Jupyter notebooks used to be called "iPython notebooks", and you will often still hear the old name. But the system has been extended to allow interfaces with other languages also, if suitable *kernels* are installed. See `https://github.com/jupyter/jupyter/wiki/Jupyter-kernels` for a list. The Python legacy lives on in the curious spelling of "Jupyter", referring to the languages Julia, Python and R.

1.2 Importing CSV files into Python

Although there are several ways to import and use CSV files in Python, the data science community uses `pandas` as a Python equivalent of R's data frames. We will do the same exclusively through this module. We will tend to use it in conjunction with `numpy`, which allows data to be worked with in a vectorised manner, which means that calculations can be done more quickly, and larger datasets can be handled. For visualisation and plotting of data frames, we will use `seaborn`, based on `matplotlib`, and use `matplotlib` for plotting more general arrays (often the outputs of machine learning procedures).

We start by importing the appropriate packages, and you should generally import all of these at the start of your session/program:

```
%matplotlib inline
import numpy as np
```

```
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns
```

Whenever you see `pd`, that refers to `pandas` etc.

Hint. If you've ever used R, you will know that it is sensible to write your commands in a text file and copy them into the R session. The same is true for Jupyter notebooks. So store all the five lines above in a text file, and copy them for every machine learning session in this module! In any particular session, we'll add further modules too. For reasons we will cover in the next lab sheet, I also have the line

```
plt.style.use("seaborn")
```

in my files (this forces `matplotlib` plots to be in the same style as the `seaborn` plots); I may switch this to `plt.style.use("ggplot")` as an alternative as it sometimes looks nicer.

`https://pandas.pydata.org/pandas-docs/stable/user_guide/10min.html` gives a nice introduction to `pandas`, but I suspect that the title "10 minutes to `pandas`" underestimates the time you should use to read this! You can always refer back to this as a useful cheat sheet for the package. For the moment, we won't need to consider `numpy` (which ensures that Python treats collections of numbers in a vectorised way); `matplotlib` and `seaborn` are the plotting packages we will consider more in the next lab sheet.

In this sheet, though, we won't bother plotting anything, so don't actually need to `import` any of those packages, except `pandas` and `numpy`, which it uses.

Let's import the `airpoll.csv` file to a data frame `airpoll`:

```
import pandas as pd
airpoll=pd.read_csv("C://Users/pm1afj/Desktop/airpoll.csv")
```

(or wherever you have saved the CSV file; if it is in the working directory, you need type only `pd.read_csv("airpoll.csv")`). Now simply type `airpoll`, and you see that the file is stored. In this case, you see that the original file has a header line, and this is assumed by `pd.read_csv`. If there is no header, one would type `pd.read_csv(.., header=None)`. This provides a set of column names `0`, `1`,.... Alternatively, you could read it in and provide headers, with `pd.read_csv(.., names=["City",..,"Mortality"])`. One should do an internet search for "`Python pandas read_csv`" to get more on the syntax.

You will notice that there is a numbered column corresponding to the row number (in Python, numbering always starts from 0), and that the column name corresponding to the cities is "`City`". Sometimes it might be preferable to have this column as the row name, although this is not generally good practice. So we won't do that, but for that, you could type

```
airpoll=pd.read_csv("C://Users/pm1afj/Desktop/airpoll.csv",index_col="City")
```

We can access the row names in `airpoll` with

```
rownames=airpoll.index
```

and the column names with

```
colnames=airpoll.columns
```

1.3 Hacking data frames in Python

First try `airpoll.shape`. What does that give you?

Let's play around with the data frame.

```
airpoll["NOX"]
```

produces all the `NOX` entries from the table.

Most machine learning techniques we will consider in this module involve *supervised learning*, where we want to understand from data how one variable depends on the others. Let's suppose we are interested in how `Mortality` depends on the other numerical variables. Then we would do something like the following:

```
features=["Rainfall","Education","Popden","Nonwhite","NOX","SO2"]
X=airpoll[features]
y=airpoll["Mortality"]
```

Type `X` to see what it contains, and `y` to see what it contains. This separation of a data frame into the "feature" variables (in `X`) and the "target" variable (in `y`) is very common. It is the standard convention for `scikit-learn`, and this is sufficiently dominant in machine learning in Python that using `X` and `y` for the features and target has become the convention for every new ML package. (Statistics traditionally uses other terminology: *input* variables, and the *response* or *output* variable.)

Typing

```
airpoll["SO2"]>100
```

produces a *Boolean* list (i.e., a list of `True` or `False`).

This data set is small enough that viewing it all is not a problem. Often, though, data sets may have too many entries and viewing the whole set is not desirable. We can type

```
airpoll.head(4)
```

to get the first 4 entries of the data.

We can select entries more generally using `iloc` if we describe them by their integer locations. So you should try

```
airpoll.iloc[3,5]
```

for the entry in row 3 and column 5 (recall that Python starts counting at 0!). Check this with `airpoll.iloc[0,0]`. We can use ranges or lists for the entries too:

```
airpoll.iloc[[1,3],2:6]
```

produces the entries for rows 1 and 3 and columns 2–5 (remember that the Python range `a:b` starts with `a` and increases to `b`, but *doesn't include* `b`). Simply typing

```
airpoll.iloc[3]
```

produces the complete entry for row 3.

There's a similar way to isolate entries, using `loc` in place of `iloc`, when we want to identify the entries by their labels. We can combine this with the Boolean lists; typing

```
airpoll.loc[airpoll[,"SO2"]>100]
```

gives all cities whose `SO2` is over 100.

Read the "10 minutes..." page above for more details on selection and operations within `pandas` data frames.

1.4 Summary statistics

Assuming we have a data frame `df`, we can find the summary statistics of every variable with

```
df.describe()
```

By default, only numeric variables are treated, but there are options you can include to get all the variables, or some specified subset. So

```
df.describe(include="all")
```

will give descriptions of each variable in `df`. Helpfully (unlike R's `summary` below), the standard deviations of each variable are also given by `describe()`.

Find the summary statistics of the `airpoll` dataframe.

A very useful computation is that of the variance matrix, which is given by `airpoll.cov()` (and the correlation matrix is given by `airpoll.corr()`). Which other variable is `Mortality` best correlated with?

One could write much more here about this; look at the documentation online or the cheat sheet for more.

1.5 Exporting CSV files from Python

If you have finished hacking your data frame, you might want to store it again as a CSV file. For this, we use

```
airpoll.to_csv("C://Users/pm1afj/Desktop/airpoll.csv")
```

(or wherever you want to store it). Again, there are various options, which allow us to change the separator to a different character etc., and you can find more of the syntax by searching "`Python pandas to_csv`".

2 Lab Sheet 1b: Handling Files in R

2.1 Notebooks

It is less common to use Jupyter notebooks for R, but it is possible, and it is not unreasonable to do it if you wish. Read the information under Python about Jupyter notebooks, and then do an internet search for "R Jupyter", say (you may want to go through the installation instructions for Python to get access to the Jupyter notebooks). There is a format from RStudio called "R Markdown notebook", which reflects their interest in "reproducible research", and which is a better alternative. In the Python section, I mentioned writing and storing script files; the RStudio interface to R does this, in addition to providing notebooks.

2.2 Importing CSV files into R

There is a file `airpoll.csv`, on Blackboard which we will use in this Lab Sheet. Perhaps the best way to do this is to get the script file `1b.R` from Blackboard (along with the data set), and save them somewhere convenient.

Let's open a new RStudio session. It is worth discussing the RStudio interface in a litte detail. When opened, there are three sections: the console, on the left, where you can type commands, and get the results of your analysis; plots and some other output (e.g., from help commands) appear in the bottom right, and the top right contains the information about what data and variables are in R's memory.

From the `File` tab, go to "Open File...", and navigate to where you stored `1b.R`, and open it. You will see that the left-hand side of the window splits into two, with the console going to the bottom left and the script file opening in the top left. So script files (just plain text files containing R commands) can be loaded into R, lines can be copied into the bottom left window and run there (simply highlight the relevant lines, and click Ctrl-Enter), with the output appearing there, or in the bottom right.

Try the top line of the script file: click somewhere on the line, and press Ctrl-Enter to copy it to the "Console" in the bottom left, and press the same again to get the next line into the console, so that they should run. In principle, the top two lines ought to load up the file, and print out the contents.

Probably they won't! Let's look at this with a little more care. R has a "working directory", and it is unlikely that this is where you put your files. Either you can change your working directory to the given one, or you can load everything in by using the full path to the file. You can see what the current working directory is by typing `getwd()`, and can change it with `setwd()` (perhaps to your desktop, or to a directory you store your data sets). If the file is in another location, you can read in the full path, with the option `file="C://users/user/Desktop/airpoll.csv"` or something. The next option, `header=TRUE`, is appropriate if your data set contains a line of titles describing the data. Notice that our file does (see the first lines above); otherwise, use `header=FALSE`. Finally, `sep=","` just ensures that the separating character is a comma (this is the default, so you can omit this). I've done the first in the script file, but it's probably better practice to do the second.

You will need to change the line `setwd(..)` *so that it is the directory where you put the data file.*

Now try the top two lines again!

You will see that to read the file into R, with the name `airpoll`, simply type

```
airpoll<-read.csv(file="airpoll.csv",header=TRUE)
```

Typing

```
airpoll
```

from within R displays the full contents of the data set, written in the form of an R *data frame*. (Let's remark that in the "tidyverse", there's a newer version of the data frame notion, called a "tibble", which has some better properties, but is otherwise pretty similar; we'll stick to the older "data frame" notion.)

Now search on the internet for "R read.csv", and find the manual pages. These are hosted within the site `https://stat.ethz.ch/` (if you are using RStudio, you can simply type `?read.csv`).

You will notice that there is a numbered column corresponding to the row number (in R, numbering always starts from 1), and that the column name corresponding to the cities is "`City`". Sometimes it might be preferable to have this column as the row name, although this is not generally good practice, and we won't do that. We can access the row names in `airpoll` with

```
airpoll_rownames<-rownames(airpoll)
```

and the column names with

```
airpoll_colnames<-colnames(airpoll)
```

2.3 Hacking data frames in R

Now try `dim(airpoll)`. What does that give you?

Let's play around with the data frame.

```
airpoll[,"NOX"]
```

produces all the `NOX` entries from the table.

Most machine learning techniques we will consider in this module involve *supervised learning*, where we want to understand from data how one variable depends on the others. Let's suppose we are interested in how `Mortality` depends on the other numerical variables. Unlike Python, different libraries in R work in different ways, and perhaps it's best to take all the numerical variables, with something like:

```
airpoll_numerical<-airpoll[,2:8]
```

Type `airpoll_numerical` to check that this works. Some libraries might follow Python, and separate out the *feature* or *input* variables (`X<-airpoll[,2:7]`) and the *target* or *output* or *response* variable (`y<-airpoll[,8]`).

Typing

```
airpoll[,"SO2"]>100
```

produces a *Boolean* list (i.e., a list of `TRUE` or `FALSE`).

This data set is small enough that viewing it all is not a problem. Often, though, data sets may have too many entries and viewing the whole set is not desirable. We can type

```
head(airpoll)
```

to get the first handful of entries of the data.

We can select entries more generally using numerical indices. So you should try:

```
airpoll[4,6]
```

for the entry in row 4 and column 6. The top left entry in the data frame is `airpoll[1,1]`. We can use ranges or lists for the entries too:

```
airpoll[c(2,4),c(3:6)]
```

produces the entries for rows 2 and 4 and columns 3–6. Typing

```
airpoll[4,]
```

produces the complete entry for row 4.

We can combine this with the Boolean lists; typing

```
airpoll[airpoll[,"SO2"]>100,]
```

gives all cities whose `SO2` is over 100.

We've done some rather limited hacking here, but much more is possible. I recommend the RStudio package `dplyr` for much more functionality. This is part of RStudio's `tidyverse`, which you hopefully installed as part of the first lab sheet; the cheat sheet on Blackboard for `tidyverse` lists a few useful functions that you can use from the `dplyr` package.

2.4 Summary statistics

In R, there is a `summary()` command, whose output gives the summary statistics of each variable in the data frame. Type

```
summary(airpoll)
```

to check. Since the city names are still in the first column, it might look better done with

```
summary(airpoll_numerical)
```

using what we did in the previous section.

For univariate data, the command `var` produces the variance of the data, and `sd` produces the standard deviation; these quantities are generally sufficient to get a good picture of the shape of the data, and it is a good first step also for multivariate data to try `var(airpoll)` to get the variance matrix (again, you might want not to have the city name involved!). Incidentally, the command `mean` produces an overall mean; better is to use the command `apply(dataframe,2,mean)` to find the mean of each variable. Try `help(apply)` for more details: the "`2`" in the command here means that the `mean` operation is applied to the *columns*. In the same way, `apply(airpoll_numerical,2,sd)^2` will give all the variances of each variable (but not the full variance matrix).

The command `cor(airpoll)` produces the correlation matrix. Which other variable is `Mortality` best correlated with?

We'll use the variance matrix a lot, but the list of individual variances (or standard deviations) is quite a nice guide to the spread of each individual variable.

2.5 Exporting CSV files from R

Internally, R stores our data `airpoll.csv` as a data frame. If you are exclusively using R, and especially if you are working with large data sets, the conversion between CSV and native R format may take some time, and it is probably more convenient to store it as a R data frame natively, rather than converting it to CSV each time. So we will save the result as an R data frame, and then every time subsequently that we open it, we will get the layout that we prefer. For this, we use the `save` command, to save the data set in R's native `.Rdata` format:

```
save(airpoll,file="airpoll.Rdata")
```

If you have changed the CSV file, as we did above, you may also want to export it back to CSV, to open in Excel (or Python) etc. For this, assuming the data frame is called `airpoll`, type

```
write.csv(airpoll,file="airpoll.csv")
```

3 Lab Sheet 1: Handling Files – the future?

It takes a little time to import CSV files into Python and R, and there is some ongoing work, spearheaded by Wes McKinney (who wrote the `pandas` package for Python) and Hadley Wickham (who wrote the `ggplot2` package for R, amongst others), to create a common file format for datasets which can be read much more quickly by Python and R, especially so that data sets can be exchanged quickly between the two programs, which are the ones used by a great majority of data scientists. McKinney's blog page, at `https://github.com/wesm` contains progress reports on these formats, called `arrow` and `feather`.