Paper DH13

SAS v. R vs. Python – A Comparison of Data Exploration Techniques

Nathanael Cockburn, Quanticate, Wilmslow, UK

ABSTRACT

There are many comparisons online of SAS vs. Python or SAS vs. R, but I find that a lot of them suffer from two issues. Firstly, they are often comparing a task that one language was designed for, while the other wasn't; an interesting comparison, but with an obvious conclusion. Secondly, in our day to day work, we generally don't have free choice of what language to use for a particular deliverable, due to either client or company requirements.

This paper will look at a scenario where that choice is both fair and plausible… initial exploration of a new dataset. I will look at what information each of the three languages can pull from a dataset in less than fifteen minutes, in order to give an overview of the data before programming on the actual deliverables begins. I will concentrate mainly, though not exclusively, on Base SAS®, Pandas (Python®) and Tidyverse (R®) as the basic modules for data exploration.

INTRODUCTION

As presented in the abstract, this paper will examine how to use the three languages to explore a new dataset. This will be a simple exploration to gain understanding of the contents; it will not include rigorous analyses or anything that would be presented in an official capacity. The techniques included could be used by anyone, from a professional data scientist to a home hobbyist.

The data exploration approach for this paper includes 5 parts:

- Reading in the data
- Checking the import
- Reviewing basic metadata
- Summary of data contents
- Exploring data trends

Each of these parts will be expanded upon below.

Technical notes: For the purposes of demonstration, the paper will use the cars.csv file that is based on the CARS dataset in the SASHELP library. All the SAS code was written using SAS University Edition; the R code uses the Tidyverse and Skimr packages; the Python code uses the Pandas module.

READING IN THE DATA

Assuming that the data is not already in a suitable format, it will need to be read into each system. Each language has straightforward import options available.

SAS – PROC IMPORT

SAS can read in most basic data files using `proc import`.

```
proc import datafile = "/folders/myfolders/data/cars.csv" out = cars;
  guessingrows = 32767;
run;
```

The `guessingrows` option needs to be included to ensure that SAS checks all the observations when setting the lengths of character variables, but otherwise the import can be run on default options. If the file is not in a CSV format, there are options to manage this; `dbms=excel` can handle Excel spreadsheets, `dbms=dlm` can support delimiters other than commas.

SAS can also read data in using a datastep and input statement. This method allows a lot more control, but for a fast view, proc import rarely causes any issues.

R – READ.CSV

R can read in CSV files using `read.csv` as a simple input method.

```
cars <- read.csv("C:/Users/Nathanael/Documents/R/cars.csv")
```

There is little customisation needed for a quick exploration. If the file uses a delimiter other than commas, the option `sep = ';'` can be added into the function, substituting whatever delimiter is needed.

PYTHON – READ_CSV

Python has a very similar option to R for simple data imports.

```
cars = pandas.read_csv("C:/Users/Nathanael/Documents/Python/cars.csv")
```

`Read_csv` has a lot of options to tweak specific elements of the import, but for a quick read, it can run on the defaults. The `sep=` option allows for different separators, in a similar way to R.

CHECKING THE IMPORT

After reading in the data, it is important to take a direct look at it and make sure that it has imported correctly; no obvious truncations, values appear to match the variables, all observations imported, etc…

A quick visual look is also the first step to seeing what the data contains – in the case of the `cars` data, it is immediately apparent that it contains information on car models, with one model per observation.

SAS – DATASET VIEWER

SAS includes the option to open a dataset directly and scroll through it in its entirety. This allows for fast verification that that data looks correct. SAS also includes filtering options for manual exploration if needed.

	Make	Model	Type	Origin	DriveTrain	MSRP	Inv
1	Acura	MDX	SUV	Asia	All	$36,945	$33
2	Acura	RSX Type S 2dr	Sedan	Asia	Front	$23,820	$21
3	Acura	TSX 4dr	Sedan	Asia	Front	$26,990	$24
4	Acura	TL 4dr	Sedan	Asia	Front	$33,195	$30
5	Acura	3.5 RL 4dr	Sedan	Asia	Front	$43,755	$39
6	Acura	3.5 RL w/Navigation 4dr	Sedan	Asia	Front	$46,100	$41
7	Acura	NSX coupe 2dr manual S	Sports	Asia	Rear	$89,765	$79
8	Audi	A4 1.8T 4dr	Sedan	Europe	Front	$25,940	$23
9	Audi	A41.8T convertible 2dr	Sedan	Europe	Front	$35,940	$32
10	Audi	A4 3.0 4dr	Sedan	Europe	Front	$31,840	$28
11	Audi	A4 3.0 Quattro 4dr manual	Sedan	Europe	All	$33,430	$30
12	Audi	A4 3.0 Quattro 4dr auto	Sedan	Europe	All	$34,480	$31
13	Audi	A6 3.0 4dr	Sedan	Europe	Front	$36,640	$33

R – DATASET VIEWER AND HEAD

R Studio has a similar dataset viewer to SAS that can be used for a quick scroll through the data.

	Make	Model	Type	Origin	DriveTrain	MSRP	Invoice	EngineSize	Cylinders	Hors
1	Acura	MDX	SUV	Asia	All	36945	33337	3.5	6	265
2	Acura	RSX Type S 2dr	Sedan	Asia	Front	23820	21761	2.0	4	200
3	Acura	TSX 4dr	Sedan	Asia	Front	26990	24647	2.4	4	200
4	Acura	TL 4dr	Sedan	Asia	Front	33195	30299	3.2	6	270
5	Acura	3.5 RL 4dr	Sedan	Asia	Front	43755	39014	3.5	6	225
6	Acura	3.5 RL w/Navigation 4dr	Sedan	Asia	Front	46100	41100	3.5	6	225
7	Acura	NSX coupe 2dr manual S	Sports	Asia	Rear	89765	79978	3.2	6	290
8	Audi	A4 1.8T 4dr	Sedan	Europe	Front	25940	23508	1.8	4	170
9	Audi	A41.8T convertible 2dr	Sedan	Europe	Front	35940	32506	1.8	4	170
10	Audi	A4 3.0 4dr	Sedan	Europe	Front	31840	28846	3.0	6	220
11	Audi	A4 3.0 Quattro 4dr manual	Sedan	Europe	All	33430	30366	3.0	6	220
12	Audi	A4 3.0 Quattro 4dr auto	Sedan	Europe	All	34480	31388	3.0	6	220
13	Audi	A6 3.0 4dr	Sedan	Europe	Front	36640	33129	3.0	6	220
14	Audi	A6 3.0 Quattro 4dr	Sedan	Europe	All	39640	35992	3.0	6	220
15	Audi	A4 3.0 convertible 2dr	Sedan	Europe	Front	42490	38325	3.0	6	220

R also has an option to view data directly in the console, using the `head()` function to print the first few observations. This is a snapshot, rather than an interactive view, but is sufficiently for quick checks. The `tail()` function can be used to print the last observations.

```
> head(cars)
   Make                Model  Type Origin DriveTrain  MSRP Invoice EngineSize Cylinders Horsepower MPG_City MPG_Highway Weight Wheelbase
1 Acura                  MDX   SUV   Asia        All 36945   33337        3.5         6        265       17          23   4451       106
2 Acura          RSX Type S 2dr Sedan   Asia      Front 23820   21761        2.0         4        200       24          31   2778       101
3 Acura               TSX 4dr Sedan   Asia      Front 26990   24647        2.4         4        200       22          29   3230       105
4 Acura                TL 4dr Sedan   Asia      Front 33195   30299        3.2         6        270       20          28   3575       108
5 Acura             3.5 RL 4dr Sedan   Asia      Front 43755   39014        3.5         6        225       18          24   3880       115
6 Acura 3.5 RL w/Navigation 4dr Sedan   Asia      Front 46100   41100        3.5         6        225       18          24   3893       115
```

PYTHON - HEAD

Python does not have a native dataset viewer (as far as this author is aware), so data must be viewed in the console. The Python console does not wrap over multiple lines, so some additional intervention may be needed to prevent columns being dropped.

```
>>> print(cars.head(5))
    Make             Model   Type  ...  Weight Wheelbase  Length
0  Acura               MDX    SUV  ...  4451.0     106.0   189.0
1  Acura  RSX Type S 2dr  Sedan  ...  2778.0     101.0   172.0
2  Acura          TSX 4dr  Sedan  ...  3230.0     105.0   183.0
3  Acura           TL 4dr  Sedan  ...  3575.0     108.0   186.0
4  Acura       3.5 RL 4dr  Sedan  ...  3880.0     115.0   197.0
```

The `iloc` function can be used to select a subset of columns and get around this issue, though this would need running multiple times to view the entire dataset.

```
>>> print(cars.iloc[:,0:7].head(5))
    Make             Model   Type Origin DriveTrain     MSRP  Invoice
0  Acura               MDX    SUV   Asia        All  36945.0  33337.0
1  Acura  RSX Type S 2dr  Sedan   Asia      Front  23820.0  21761.0
2  Acura          TSX 4dr  Sedan   Asia      Front  26990.0  24647.0
3  Acura           TL 4dr  Sedan   Asia      Front  33195.0  30299.0
4  Acura       3.5 RL 4dr  Sedan   Asia      Front  43755.0  39014.0
```

Also of note, unlike R, Python doesn't print output by default and the `print()` function must always be deployed to see any results.

REVIEWING THE METADATA

After confirming that the import was successful, any further exploration is going to require some knowledge of the metadata. The most important information for subsequent steps will be variable names and which variables are character vs. numeric. However, lengths of character variables, SAS formats (if the data came as a .sas7bdat file) and other metadata may be helpful too.

SAS – SASHELP.VCOLUMN AND PROC CONTENTS

SAS has two potential routes to viewing metadata for a dataset. The SASHELP library, which is automatically created at the start of a SAS session, contains two datasets, `sashelp.vtable` and `sashelp.vcolumn`; `sashelp.vtable` contains dataset level information, while `sashelp.vcolumn` contains variable information. These datasets contain information on every SAS dataset in every library, so need filtering to find the `cars` dataset. The `sashelp.vcolumn` dataset, partially displayed below, contains variable names, types, lengths and formats.

	libname	memname	memtype	name	type	length
1	WORK	CARS	DATA	Make	char	13
2	WORK	CARS	DATA	Model	char	39
3	WORK	CARS	DATA	Type	char	6
4	WORK	CARS	DATA	Origin	char	6
5	WORK	CARS	DATA	DriveTrain	char	5
6	WORK	CARS	DATA	MSRP	num	8
7	WORK	CARS	DATA	Invoice	num	8
8	WORK	CARS	DATA	EngineSize	num	8
9	WORK	CARS	DATA	Cylinders	num	8
10	WORK	CARS	DATA	Horsepower	num	8
11	WORK	CARS	DATA	MPG_City	num	8
12	WORK	CARS	DATA	MPG_Highway	num	8
13	WORK	CARS	DATA	Weight	num	8
14	WORK	CARS	DATA	Wheelbase	num	8
15	WORK	CARS	DATA	Length	num	8

The other option for viewing SAS variable information is `proc contents`, which will output less information than is contained in `sashelp.vcolumn`, but may be quicker if only the basic information is required.

```
proc contents data = cars;
run;
```

Alphabetic List of Variables and Attributes					
#	Variable	Type	Len	Format	Informat
9	Cylinders	Num	8	BEST12.	BEST32.
5	DriveTrain	Char	5	$5.	$5.
8	EngineSize	Num	8	BEST12.	BEST32.
10	Horsepower	Num	8	BEST12.	BEST32.
7	Invoice	Num	8	BEST12.	BEST32.
15	Length	Num	8	BEST12.	BEST32.
11	MPG_City	Num	8	BEST12.	BEST32.
12	MPG_Highway	Num	8	BEST12.	BEST32.
6	MSRP	Num	8	BEST12.	BEST32.
1	Make	Char	13	$13.	$13.
2	Model	Char	39	$39.	$39.
4	Origin	Char	6	$6.	$6.
3	Type	Char	6	$6.	$6.
13	Weight	Num	8	BEST12.	BEST32.
14	Wheelbase	Num	8	BEST12.	BEST32.

R – STR

R has the `str()` function that can be run on a dataset to provide basic information. It provides a little more insight than SAS, as it hints towards the structure of the data as well, rather than just the variable information.

```
> str(cars)
'data.frame':   428 obs. of  15 variables:
 $ Make       : Factor w/ 38 levels "Acura","Audi",..: 1 1 1 1 1 1 1 2 2 2 ...
 $ Model      : Factor w/ 425 levels "3.5 RL 4dr","3.5 RL w/Navigation 4dr",..: 245 309 388 3
 $ Type       : Factor w/ 6 levels "Hybrid","Sedan",..: 4 2 2 2 2 2 3 2 2 2 ...
 $ Origin     : Factor w/ 3 levels "Asia","Europe",..: 1 1 1 1 1 1 1 2 2 2 ...
 $ DriveTrain : Factor w/ 3 levels "All","Front",..: 1 2 2 2 2 2 3 2 2 2 ...
 $ MSRP       : num  36945 23820 26990 33195 43755 ...
 $ Invoice    : num  33337 21761 24647 30299 39014 ...
 $ EngineSize : num  3.5 2 2.4 3.2 3.5 3.5 3.2 1.8 1.8 3 ...
 $ Cylinders  : num  6 4 4 6 6 6 6 4 4 6 ...
 $ Horsepower : num  265 200 200 270 225 225 290 170 170 220 ...
 $ MPG_City   : num  17 24 22 20 18 18 17 22 23 20 ...
 $ MPG_Highway: num  23 31 29 28 24 24 24 31 30 28 ...
 $ Weight     : num  4451 2778 3230 3575 3880 ...
 $ Wheelbase  : num  106 101 105 108 115 115 100 104 105 104 ...
 $ Length     : num  189 172 183 186 197 197 174 179 180 179 ...
```

From this output, the character/numeric split is seen, but also the structure of the character variables. Model has 425 unique values out of 428 observations, so is essentially a free text field, but Type, Origin and DriveTrain are clearly categorical variables with a relatively small number of unique values.

PYTHON – INFO

The `info()` function in Python performs similarly to R, though provides less structural information. It still gives the important character/numeric split (or object/float in Python).

```
>>> print(cars.info())
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 428 entries, 0 to 427
Data columns (total 15 columns):
Make            428 non-null object
Model           428 non-null object
Type            428 non-null object
Origin          428 non-null object
DriveTrain      428 non-null object
MSRP            428 non-null float64
Invoice         428 non-null float64
EngineSize      428 non-null float64
Cylinders       426 non-null float64
Horsepower      428 non-null float64
MPG_City        428 non-null float64
MPG_Highway     428 non-null float64
Weight          428 non-null float64
Wheelbase       428 non-null float64
Length          428 non-null float64
dtypes: float64(10), object(5)
```

SUMMARY OF DATA CONTENTS

Armed with the general structure of the data, the next step is to look at some basic summaries.

SAS – FREQ AND MEANS

SAS generally summarises numeric variables and character variables via different routes. There are various options available, but the quickest options for standard summaries are `proc freq` and `proc means`.

Proc freq is used to provide frequency counts, typically for character variables, though a discrete numeric variable could be summarised in this way.

```
proc freq data = cars (keep=make type origin drivetrain);
run;
```

This code will output basic frequency counts for all variables in the dataset. The keep statement is a simple way to pick out the character variables of interest; Model is not included, as the frequency count would be 1 in almost every case.

Type	Frequency	Percent	Cumulative Frequency	Cumulative Percent
Hybrid	3	0.70	3	0.70
SUV	60	14.02	63	14.72
Sedan	262	61.21	325	75.93
Sports	49	11.45	374	87.38
Truck	24	5.61	398	92.99
Wagon	30	7.01	428	100.00

Origin	Frequency	Percent	Cumulative Frequency	Cumulative Percent
Asia	158	36.92	158	36.92
Europe	123	28.74	281	65.65
USA	147	34.35	428	100.00

These summaries show some immediate patterns (or lack of them). Sedans account for the majority of the cars in the dataset, the origin is very evenly split; similar observations can be made on the `make` and `drivetrain` variables.
`Proc freq` has a lot of customisation options. Introducing a by statement would start to nest some of the frequencies (e.g. summarising type within each individual origin). It is also possible to output information as a SAS dataset, allowing further analysis, beyond the scope of this initial exploration.

While proc freq summarises character data, proc means can be used to summarise the numeric variables. In a similar fashion to proc freq, it can be run without any options to create simple summary stats as a starting point, again keeping only some variables of interest.

```
proc means data = cars (keep=cylinders horsepower weight);
run;
```

The summary stats produced are very basic, but allow for some observations.

Variable	N	Mean	Std Dev	Minimum	Maximum
Cylinders	426	5.8075117	1.5584426	3.0000000	12.0000000
Horsepower	428	215.8855140	71.8360316	73.0000000	500.0000000
Weight	428	3577.95	758.9832146	1850.00	7190.00

There are two missing values for `cylinders` immediately visible. Additionally, with values ranging from 3 to 12, `cylinders` looks like it might benefit from a frequency count, as much as these summary stats.
The mean for `horsepower` is lower than the midpoint between min and max, suggesting that the distribution skews towards the lower end. A median value would provide extra insight here – this is not one of the default stats, but `proc means` can be customised to include a wide range of stats. For example, the code below will output percentiles of 10, 25, 50, 75, and 90 for the variable `horsepower`.

```
proc means data = cars n p10 p25 median p75 p90;
  var horsepower;
run;
```

Analysis Variable : Horsepower					
N	10th Pctl	25th Pctl	Median	75th Pctl	90th Pctl
428	130.0000000	165.0000000	210.0000000	255.0000000	302.0000000

R – SUMMARY AND SKIMR
Unlike SAS, R has a single function to summarise character and numeric data at the same time, `summary()`.

```
> summary(cars)
        Make                       Model        Type        Origin    DriveTrain       MSRP          Invoice        EngineSize
Toyota      : 28   C240 4dr             :  2   Hybrid:  3   Asia  :158   All  : 92   Min.   : 10280   Min.   :  9875   Min.   :1.300
Chevrolet   : 27   C320 4dr             :  2   Sedan :262   Europe:123   Front:226   1st Qu.: 20334   1st Qu.: 18866   1st Qu.:2.375
Mercedes-Benz: 26   G35 4dr              :  2   Sports: 49   USA   :147   Rear :110   Median : 27635   Median : 25295   Median :3.000
Ford        : 23   3.5 RL 4dr           :  1   SUV   : 60                           Mean   : 32775   Mean   : 30015   Mean   :3.197
BMW         : 20   3.5 RL w/Navigation 4dr:  1   Truck : 24                           3rd Qu.: 39205   3rd Qu.: 35710   3rd Qu.:3.900
Audi        : 19   300M 4dr             :  1   Wagon : 30                           Max.   :192465   Max.   :173560   Max.   :8.300
(Other)     :285   (Other)              :419
  Cylinders        Horsepower      MPG_City      MPG_Highway       Weight       Wheelbase        Length
Min.   : 3.000   Min.   : 73.0   Min.   :10.00   Min.   :12.00   Min.   :1850   Min.   : 89.0   Min.   :143.0
1st Qu.: 4.000   1st Qu.:165.0   1st Qu.:17.00   1st Qu.:24.00   1st Qu.:3104   1st Qu.:103.0   1st Qu.:178.0
Median : 6.000   Median :210.0   Median :19.00   Median :26.00   Median :3474   Median :107.0   Median :187.0
Mean   : 5.808   Mean   :215.9   Mean   :20.06   Mean   :26.84   Mean   :3578   Mean   :108.2   Mean   :186.4
3rd Qu.: 6.000   3rd Qu.:255.0   3rd Qu.:21.25   3rd Qu.:29.00   3rd Qu.:3978   3rd Qu.:112.0   3rd Qu.:194.0
Max.   :12.000   Max.   :500.0   Max.   :60.00   Max.   :66.00   Max.   :7190   Max.   :144.0   Max.   :238.0
NA's   :2
```

This function provides identical frequency counts to SAS for the characters variables, albeit truncating the lists for `make` and `model` that have a lot of unique values. The full frequency counts could be produced for these variables using the function `table(cars$make)`.

For numeric variables, a slightly different set of statistics is produced compared to SAS; the quartiles are included by default, but SD is not.

An alternative function for summarising in R is the `skimr()` function.

```
-- Variable type: factor -------------------------------------------------------------------------------------------
# A tibble: 5 x 6
  skim_variable n_missing complete_rate ordered n_unique top_counts
* <chr>              <int>         <dbl> <lgl>      <int> <chr>
1 Make                   0             1 FALSE         38 Toy: 28, Che: 27, Mer: 26, For: 23
2 Model                  0             1 FALSE        425 C24: 2, C32: 2, G35: 2, 3.5: 1
3 Type                   0             1 FALSE          6 Sed: 262, SUV: 60, Spo: 49, Wag: 30
4 Origin                 0             1 FALSE          3 Asi: 158, USA: 147, Eur: 123
5 DriveTrain             0             1 FALSE          3 Fro: 226, Rea: 110, All: 92

-- Variable type: numeric ------------------------------------------------------------------------------------------
# A tibble: 10 x 11
   skim_variable n_missing complete_rate     mean       sd      p0      p25      p50      p75     p100 hist
 * <chr>            <int>         <dbl>    <dbl>    <dbl>   <dbl>    <dbl>    <dbl>    <dbl>    <dbl> <chr>
 1 MSRP                 0             1    32775.   19432.   10280   20334.   27635    39205   192465
 2 Invoice              0             1    30015.   17642.    9875   18866    25294.   35710.  173560
 3 EngineSize           0             1        3.2      1.1     1.3     2.4      3        3.9      8.3
 4 Cylinders            2         0.995        5.8      1.6     3       4        6        6       12
 5 Horsepower           0             1      216.      71.8    73     165      210      255      500
 6 MPG_City             0             1       20.1      5.2    10      17       19       21.2     60
 7 MPG_Highway          0             1       26.8      5.7    12      24       26       29       66
 8 Weight               0             1     3578.     759.    1850    3104     3474.    3978.    7190
 9 Wheelbase            0             1      108.       8.3    89     103      107      112      144
10 Length               0             1      186.      14.4   143     178      187      194      238
```

The frequency counts are slightly less readable, but the statistics for numeric variables have the SD included, as well as a very basic display of the distribution.

PYTHON – DESCRIBE AND VALUE_COUNTS

Python behaves closer to SAS, as it has separate functions for summary stats and frequency counts. The `describe()` function produces the standard summary statistics by default, though suffers the same issue as `head()` in truncating the output when there isn't enough room.

```
>>> print(cars.describe())
                MSRP        Invoice  ...    Wheelbase      Length
count     428.000000     428.000000  ...   428.000000  428.000000
mean    32774.855140   30014.700935  ...   108.154206  186.362150
std     19431.716674   17642.117750  ...     8.311813   14.357991
min     10280.000000    9875.000000  ...    89.000000  143.000000
25%     20334.250000   18866.000000  ...   103.000000  178.000000
50%     27635.000000   25294.500000  ...   107.000000  187.000000
75%     39205.000000   35710.250000  ...   112.000000  194.000000
max    192465.000000  173560.000000  ...   144.000000  238.000000

[8 rows x 10 columns]
```

For the frequency counts, the function `value_counts()` is a quick option for single variables.

```
>>> print(cars["Origin"].value_counts())
Asia      158
USA       147
Europe    123
Name: Origin, dtype: int64
```

It is possible to run `value_counts()` across all the variables, using a `for` loop.

```
for var in ["Origin","DriveTrain","Type"]:
    print(cars[var].value_counts())
```

This will produce the output for each variable, though the list of variables must be specified, rather than relying on Python to select the character variables itself.

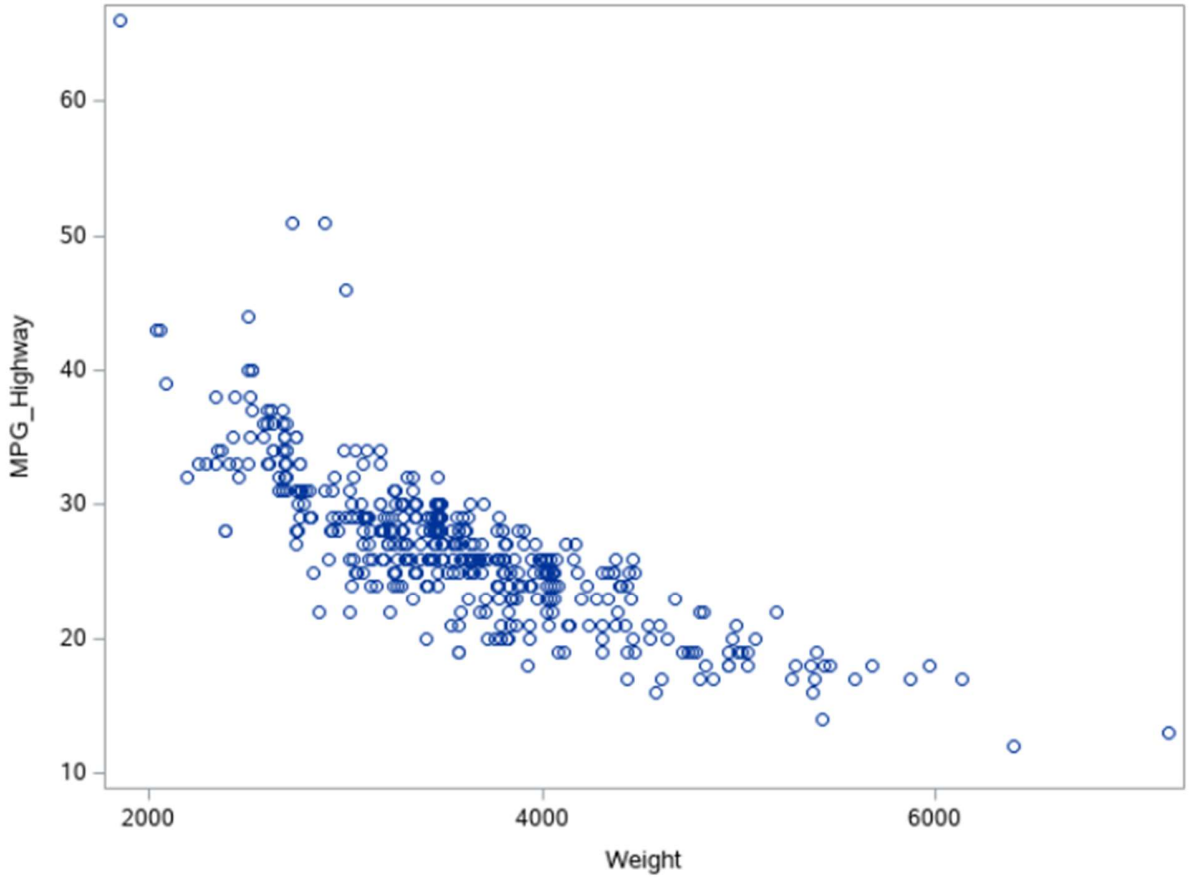
EXPLORING DATA TRENDS

The final stage of this data exploration is to identify any obvious trends and correlations between variables. In absence of any statistical analysis, these will most easily and quickly be spotted by plotting the data.

SAS – PROC SGPLOT

SAS can produce extremely customised graphics using proc sgplot, but its simplest variation is a scatter plot, where the only options needed are the X and Y variables.

```
proc sgplot data = cars;
  scatter x = weight y = mpg_highway;
run;
```



There is an obvious correlation immediately visible – heavy cars have worse fuel efficiency. However, this pair of variables was selected with the knowledge that this correlation would exist. For data that is completely new to the programmer, a comparison of every variable against every other variable would be needed to search for such trends. SAS can do this, but the code begins to get a lot more involved.

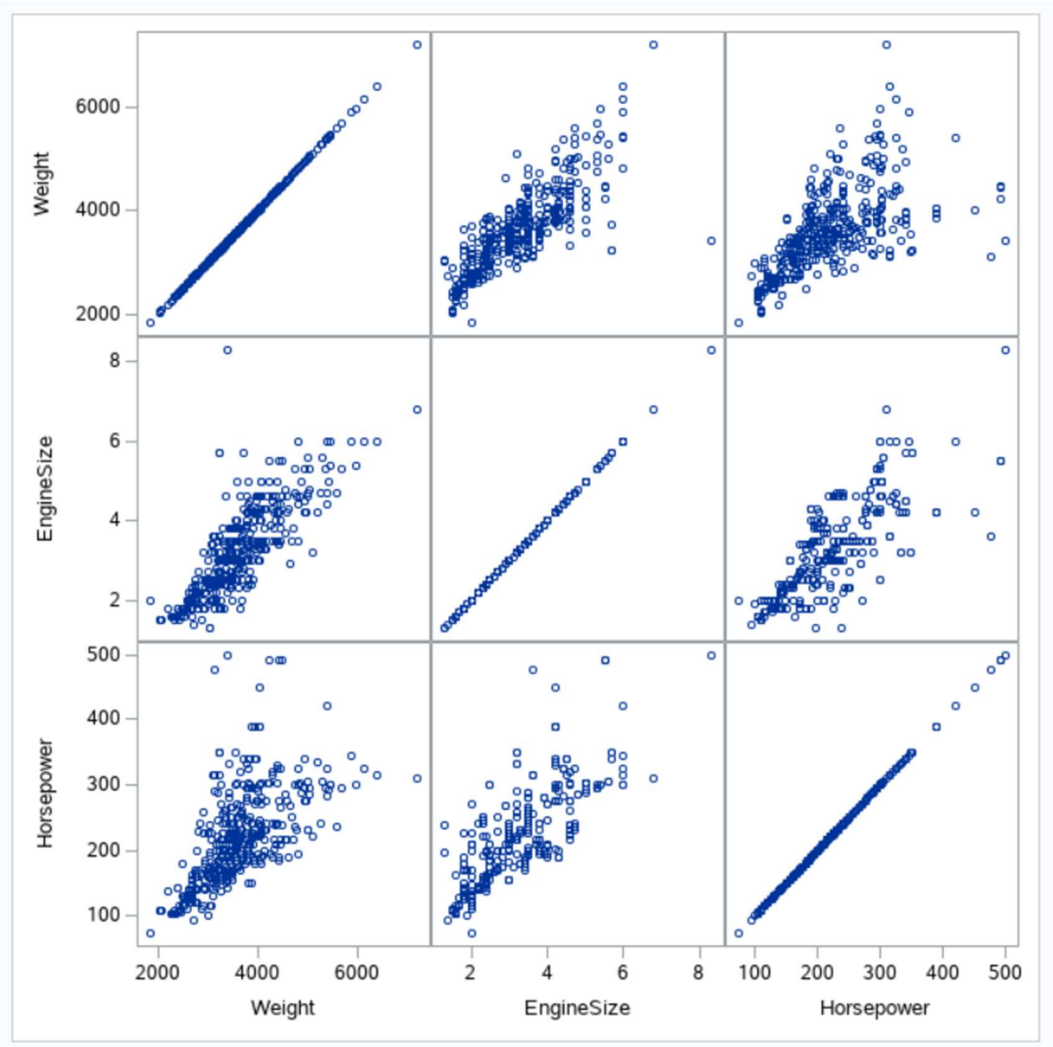
```
*Create a macro to run each plot;
%macro plot(x=,y=);
  proc sgplot data = cars;
    scatter x=&x y=&y;
  run;
%macro;

*Programmatically select all the numeric variables from the data;
proc sql noprint;
  select distinct name into :var1-:var10
  from sashelp.vcolumn
  where type = "num" and memname = "CARS";
quit;

*Run the plot macro over every variable pair;
%macro multiplot;
  %do i = 1 %to 10;
    %do j = 1 %to 10;
      %plot(x=&&var&i,y=&&var&j);
    %end;
  %end;
%mend;
%multiplot;
```

This code will produce 100 plots, with every variable against every other variable (plus itself). The result is achieved, but not in a quick fashion.
More recent versions of SAS (9.3 onwards) have a shorter way to compare variables, using `proc sgscatter` to create a trellis plot.

```
proc sgscatter data= cars;
  compare y=(weight enginesize horsepower )
          x=(weight enginesize horsepower );
run;
```

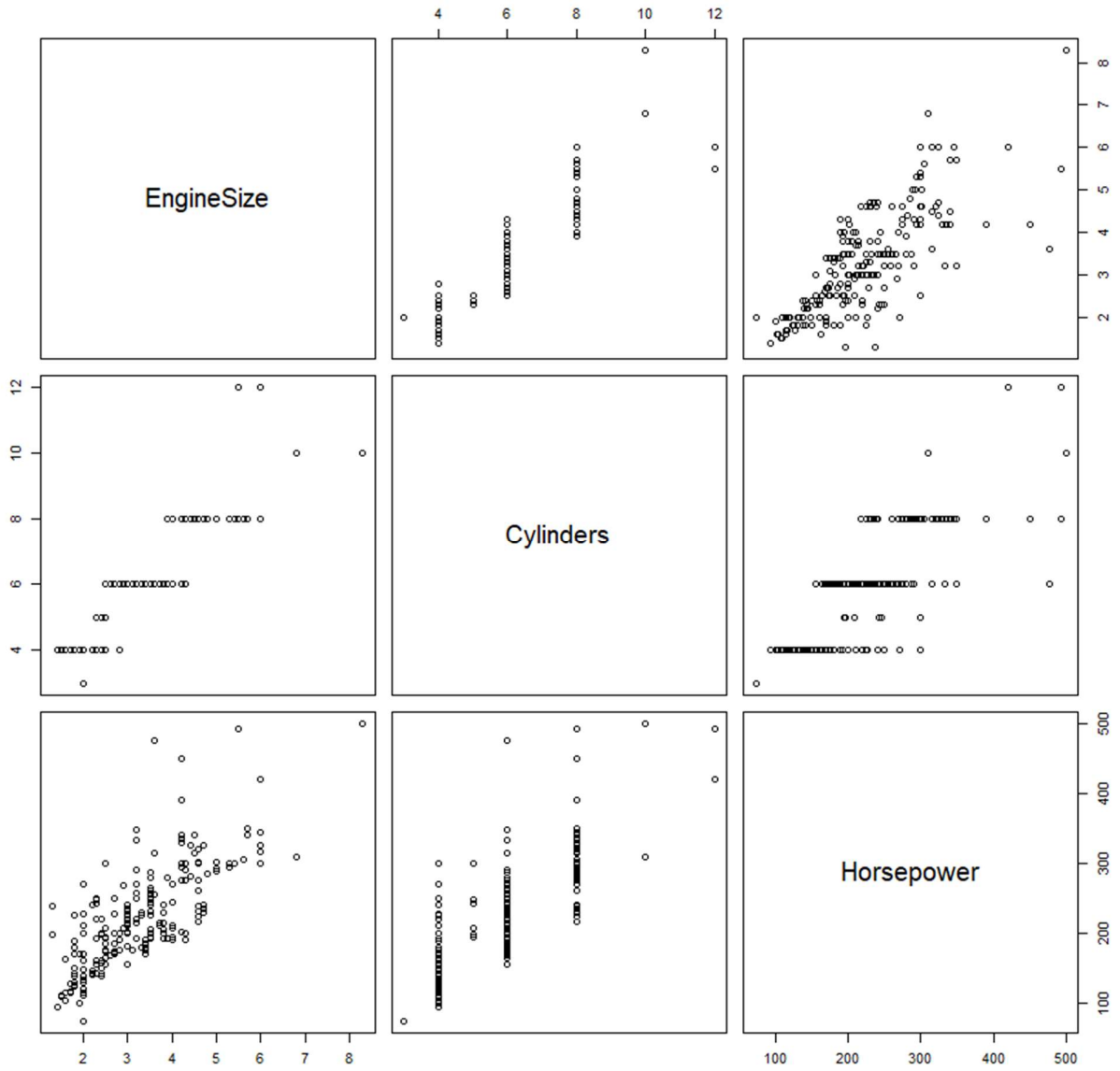


This is a much faster way to do the comparisons, though it does require specifying the variables required – it may need running several times in different combinations, depending on how many variables are on the dataset.

R – PAIRS

R has a simple option for plotting variables against each other with the `pairs()` function.

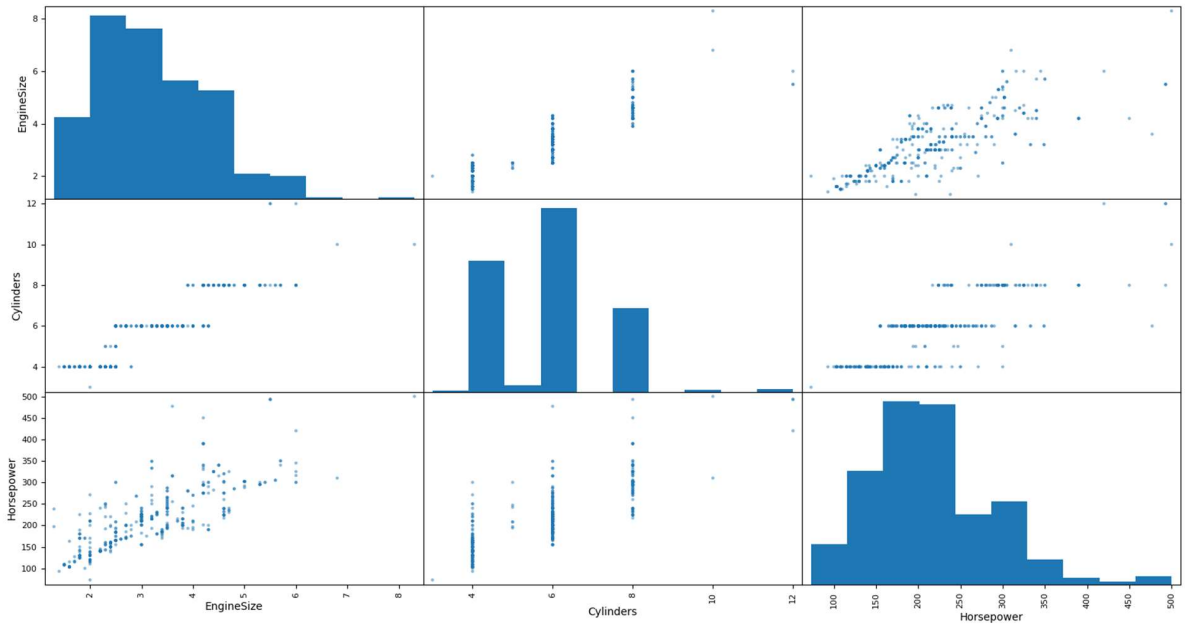
`pairs(cars)` would produce a 10 by 10 plot that would be very difficult to interpret, so it may be needed to plot a subset of variables, for example `pairs(cars[8:10])`.



Trends and correlations are immediately visible; more cylinders generally correspond to more horsepower, and similarly with engine size.
There are mains options in `pairs()` to adjust the appearance of the plot or use logarithmic axes, but for the purposes of a quick exploration, the default plots have all the information required.

PYTHON – SCATTER_MATRIX
Python has a very similar option to R for plotting a set of trellised scatter plots.

```
scatter_matrix(cars.iloc[:,7:10])
plt.show()
```



The same trends are visible as were in the R plots, but the Python scatter matrix has the added benefit of displaying the distributions on the diagonal. Again, there is a need to limit the variables being plotted – Python will plot the 10 by 10 matrix with no issues, but it would be impossible to see anything, except on the largest screens.

A NOTE ON TIDYVERSE

The original plan for this paper, as stated in the abstract, was to use Tidyverse for the R exploration sections. However, as it was being written, it became clear that the core R functions were adequate for the central conceit – what can be discovered in just 15 minutes?

Nonetheless, for anyone looking to take a step past this initial exploration phase, Tidyverse provides excellent tools to start sorting (`arrange`), subsetting (`filter`) and manipulating (`mutate`) the data in a straightforward and speedy fashion.

CONCLUSION

All three languages are capable of performing basic data exploration tasks, as shown in this paper. However, there are differences in their use that are worth noting.

SAS is very powerful and has a lot of options to customise things, but this does lead to a lot more code than the other two languages, which may be off-putting to a new programmer. In particular, SAS graphics are slow and relatively more complex to produce if the purpose is just a quick review.

Python is not a statistical programming language and this shows. Pandas does a good job of making data exploration tools more available, but they feel a little bit clunky to use; the output doesn't always format nicely in the console and the programmer has to put more thought in to get the right results.

R is very good for this kind of data dive. Simple functions provide a lot of information without needing to tweak too many options, if any. A combination of str(), summary() and pairs() can quickly give a very good overview of a dataset, in far less than 15 minutes.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Nathanael Cockburn
Quanticate
Blackbox, Beech Lane
Wilmslow, SK9 5ER
+44 1625 544834
nathanael.cockburn@quanticate.com
www.quanticate.com