Creating Documents Programatically with R Markdown

Liam Wright

31 May, 2019

Introduction

In this document, I give a brief overview of writing documents programatically using **Markdown**. Markdown is an accessible language, with a relatively shallow learning curve and lots of help available online. Its virtue lies in its compatibility with a number of statistical software languages, primarily R, but also including SAS and STATA.

With Markdown, analysts can weave statistical output directly into their document. This includes tables, graphs and even entire paragraphs, the wording of which may depend on some result (e.g. write "employment has increased" if a variable is greater than zero). This can be a vast improvement over more manual methods, such as copying and pasting results from Excel to Word or Powerpoint. Manual methods aren't reproducible and leave less of a data trail: an analyst can quickly forget how they produced the output they did, and if they need to produce similar output in a different setting (or update previous results), they often need to start again from scratch. Such a process is inefficient and error-prone. Indeed, there is a triple risk from using manual methods: errors are more likely to occur, less likely to be noticed, and less easily fixed as the source of the problem is not as easily identified.

To give a simple demonstration of producing documents with Markdown, I create a simple report on changes in employment levels by country of origin using the latest data from the Labour Force Survey. To add a further degree of automation to the report, I download this data using the **ONS API** (Application Programming Interface). I discuss what an API is in further detail below.

Markdown

In this section, I provide a slightly longer introduction to Markdown. Markdown is a 'markup' language. Unlike with Word, where "What You See is What You Get" (i.e. formatting is done on the text and immediately observed), in Markdown formatting is added to the text as code. Editing is done on a plain-text document and a compiler is used to produce a separate document with the formatting rendered correctly (e.g. a PDF or DOCX). For instance, a single `#` in the plain-text document will specify a title in the final document, while two asterisks `**` either side of some text will make it **bold**. See **stackedit.io** for an online Markdown editor and basic formatting tutorial.

RStudio contains a package called **rmarkdown** which allows users to create **R Markdown** documents. These use the Markdown language and allow users to "knit" R code and output into their documents. R can be interfaced with Stata or SAS, so output can be drawn from these programmes, too. However, to my knowledge, this will not work with RStudio Server. Alternatively, in Stata, the user-written **Markstat** package is available. This does not require RStudio. To open a new R Markdown file in RStudio click `File > New File > R Markdown`. The file which opens gives a basic tutorial on R Markdown. There is also extensive help avaible online. I do not discuss the mechanics of using R Markdown further here.

APIs

To produce my report, I download quarterly employment statistics by country of origin *via* the ONS API. Simplifying somewhat, an API is a URL. Think of a URL as an instruction to some server (e.g. the BBC) to return a specific document (a news article). Most URLs return a HTML document, which your browser then renders as a nice-looking webpage. With an API, a piece of structured text is returned instead, which is

typically in a format such as **JSON** or **XML**. The specific content of this text is determined by the **API Call** - that is, the parameters specified within the URL string - while the structure of this text follows rules specified by the API owner (and generally documented online).

To illustrate this with a simple example, the Postcodes.io API returns a JSON string with fields for different geographies a postcode sits within (e.g. LSOA, CCG) for postcodes supplied by the user in the API call. For instance, `http://api.postcodes.io/postcodes/SW1H9NA` returns geographies for Caxton House (see screenshot below). These fields have the same names as they would be if I'd put in another (valid) postcode. Output is therefore predictable, which means with APIs, an analysis can be updated by making very few changes to the underlying code, or even by running the code again when new data has been published. Used alongside Markdown, whole reports can be reproduced with the click of a button.

To make this a bit clearer, here is how the Postcodes.io API call for SW1H 9NA looks when accessed in Chrome. Notice that it is a list of key-value pairs.


```
{"status":200,"result":{"postcode":"SW1H
9NA","quality":1,"eastings":529805,"northings":179570,"country":"England","nhs_ha":"London","longitude":-0.13132,"latitude":51.5001
71,"european_electoral_region":"London","primary_care_trust":"Westminster","region":"London","lsoa":"Westminster
020A","msoa":"Westminster 020","incode":"9NA","outcode":"SW1H","parliamentary_constituency":"Cities of London and
Westminster","admin_district":"Westminster","parish":"Westminster, unparished area","admin_county":null,"admin_ward":"St
James's","ced":null,"ccg":"NHS Central London (Westminster)","nuts":"Westminster","codes":
{"admin_district":"E09000033","admin_county":"E99999999","admin_ward":"E05000644","parish":"E43000236","parliamentary_constituency"
:"E14000639","ccg":"E38000031","ced":"E99999999","nuts":"UKI32"}}}
```


Figure 1: Geographies for SW1H 9NA via Postcodes.io API

Employment by Country of Birth

I am interested in getting time series data for total employment in the UK split by country of birth and nationality (UK, EU and Non-EU). The typical method would be to find a spreadsheet(s) on the ONS website which contains this information. In this case, the relevant spreadsheet is "EMP06: Employment by country of birth and nationality" (link here). Instead, with the ONS API, I can directly download the individual time series within this spreadsheet using an API call of the following format:

`https://api.ons.gov.uk/dataset/{dataset_id}/timeseries/{timeseries_id}/data`

The relevant time series IDs are displayed in row 11 of the sheets within *EMP06*. Alternatively, I could find the time series IDs using the ONS's **time-series explorer** (link here), or by downloading metadata on available time series using the ONS API (documentation here). It isn't clear from the *EMP06* spreadsheet, but the relevant dataset ID for us is "LMS", which is short for labour market statistics. (This information can also be found via API Call.) So, to download total employment figures, I could access the website https://api.ons.gov.uk/dataset/lms/timeseries/mgtm/data

Here is how the top of this API call looks in Chrome. Notice again how this is a list of key-value pairs, with an entry for each month.


```
{"description":{"cdid":"MGTM","contact":{"email":"labour.market@ons.gov.uk","name":"Richard Clegg","telephone":"+44 (0)1633
455400"},"datasetId":"LMS","datasetUri":"/employmentandlabourmarket/peopleinwork/employmentandemployeetypes/datasets/labourmarket
statistics","date":"2019 FEB","monthLabelStyle":"three month average","nextRelease":"11 June
2019","number":"32641","preUnit":"","releaseDate":"2019-05-13T23:00:00.000Z","sampleSize":"0","source":"","title":"LFS: In
employment: UK: All: Aged 16 and over: Thousands: NSA","unit":""},"months":[{"date":"1984 APR","label":"1984 MAR-
MAY","month":"April","quarter":"","sourceDataset":"LMS","updateDate":"2015-10-13T23:00:00.000Z","value":"23974","year":"1984"},
{"date":"1984 MAY","label":"1984 APR-JUN","month":"May","quarter":"","sourceDataset":"LMS","updateDate":"2015-10-
13T23:00:00.000Z","value":"","year":"1984"},{"date":"1984 JUN","label":"1984 MAY-
JUL","month":"June","quarter":"","sourceDataset":"LMS","updateDate":"2015-10-13T23:00:00.000Z","value":"","year":"1984"},
```


Figure 2: Total UK Employment via ONS API

To access this data more programatically, in my code, I add the relevant series IDs to a dataframe with information on what each series ID represents. I am going to loop over these IDs to download each time series separately as the whole dataset can't be downloaded at once.

```
dataset_ids <- data.frame(code=c("MGTM","JF6H","EQ4U","EQ4W",
                                 "JF6F","EQ4Q","EQ4S"),
                          origin=c("Total","UK","EU",
                                   "Non-EU","UK",
                                   "EU","Non-EU"),
                          definition=c("Total",rep("National",3),
                                       rep("Country of Birth",3)),
                          stringsAsFactors=FALSE)
dataset_ids
```

```
##   code origin       definition
## 1 MGTM  Total            Total
## 2 JF6H     UK         National
## 3 EQ4U     EU         National
## 4 EQ4W Non-EU         National
## 5 JF6F     UK Country of Birth
## 6 EQ4Q     EU Country of Birth
## 7 EQ4S Non-EU Country of Birth
```

The following code shows this loop. It first creates a new URL string including the relevant series ID and then makes the API call using the function `GET`.[1] The returned data is in JSON format, with characters rendered in Unicode. The `rawToChar` function transforms the Unicode into human readable text, and the resulting JSON is parsed into a list (`fromJSON`; a list in R is a collection of data types which can be of different lengths). This list contains a lot of information that is irrelevant to us (e.g. contact details for the person responsible for the data). I just want the information from the 'quarters' object within the returned list. This object is a data frame from which I extract columns (variables) for year, quarter, and value. I reformat some of these variables into numbers as they were stored as characters originally. The last bit of code in the loop is a conditional statement to append data from each API call together. I call the resulting object `long`.

```
for (i in dataset_ids$code){
  path <- paste("dataset/LMS/timeseries/",i,"/data")
  raw <- GET(url = "https://api.ons.gov.uk",path = path)
  df <- fromJSON(rawToChar(raw$content))$quarters[c("quarter","value","year")] %>%
        mutate(quarter=substr(quarter,2,2))
  df <- as.data.frame(sapply(df,as.numeric)) %>%
        mutate(date=as.yearqtr(year+(quarter-1)/4),
               date.label=paste0(year," Q",quarter),
               code=i) %>%
        left_join(dataset_ids,by="code")
  df[c("quarter","year")] <- NULL

  if (exists("long")==TRUE) {
    long <- rbind(df,long)
  }
  else{
    long <- df
  }

}
```

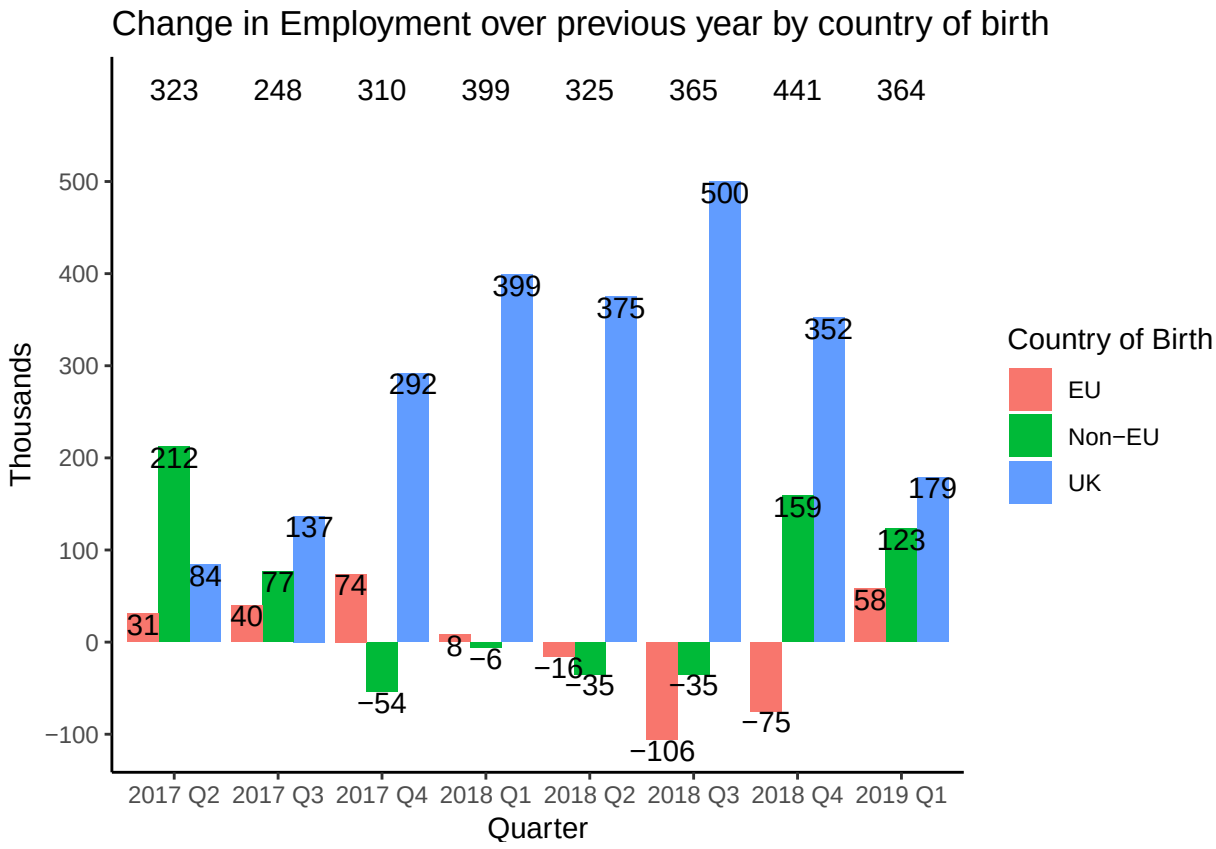
[1] `GET` is a HTTP protocol method for communicating with a server over the internet. API calls often use this or the `POST` method. See w3schools for more information on HTTP methods (link here).

The dataset has information going back as far as 1989 for each series. I am only interested in the eight most recent observations, so I subset the data using the following (probably verbose) code. I now have an object which has a row for each quarter from 2017 Q2 to 2019 Q1 for each nationality/country of birth (56 rows in total). I have it in this long format because `ggplot`, the package I will use to graph the data, works better with data in this format.

```
long <- long %>% group_by(code) %>%
  mutate(change=value-dplyr::lag(value, n = 4)) %>%
  ungroup()
long.last8 <- long[long$date %in%
                     unique(long$date)[rank(unique(-long$date))<=8],]
```

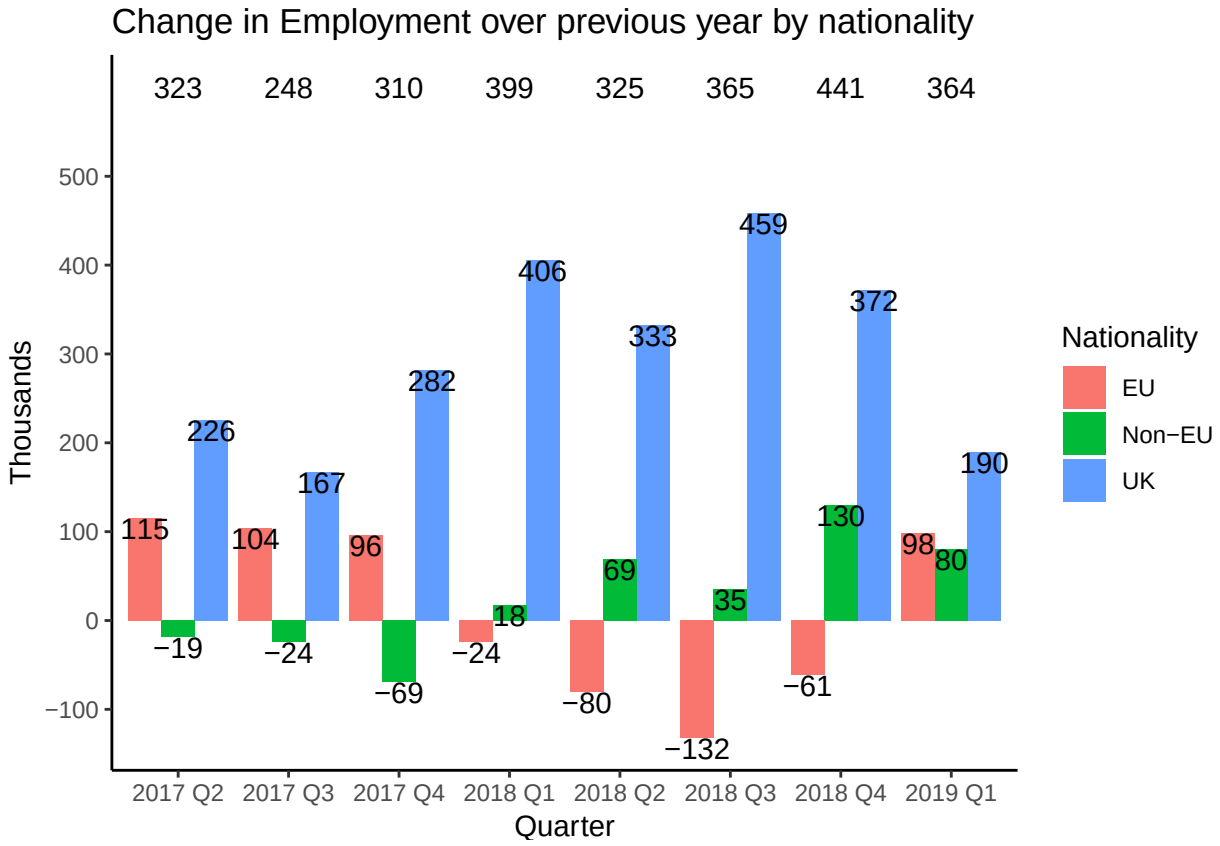
Next I plot the results. Results by county of birth:

```
plot.country <- ggplot(long.last8[long.last8$definition=="Country of Birth",],
                aes(x=date.label,y=change))+
                geom_bar(aes(fill=origin), stat="identity", position="dodge")+
                geom_text(aes(label=change,group=origin),vjust=1,
                          position = position_dodge(width=0.9))+
                geom_text(data=long.last8[long.last8$definition=="Total",],
                          aes(label=change,y=600))+
                labs(title="Change in Employment over previous year by country of birth",
                     x="Quarter",y="Thousands")+
                scale_y_continuous(breaks=-1:5*100)+
                scale_fill_discrete(name="Country of Birth")+
                theme_classic()
plot.country
```



Results by nationality:

```
plot.nationality <- ggplot(long.last8[long.last8$definition=="National",],
                    aes(x=date.label,y=change))+
                     geom_bar(aes(fill=origin), stat="identity", position="dodge")+
                     geom_text(aes(label=change,group=origin),vjust=1,
                               position = position_dodge(width=0.9))+
                     geom_text(data=long.last8[long.last8$definition=="Total",],
                               aes(label=change,y=600))+
                     labs(title="Change in Employment over previous year by nationality",
                          x="Quarter",y="Thousands")+
                     scale_y_continuous(breaks=-1:5*100)+
                     scale_fill_discrete(name="Nationality")+
                     theme_classic()
plot.nationality
```



And I finish by writing a paragraph which uses conditional statements to generate text based on the most recent data:

```
total.emp <- long[long$origin=="Total",]
total.emp <- total.emp[order(total.emp$date,decreasing=TRUE),]
total.emp.current <- total.emp$value[1]
total.emp.change <- total.emp$change[1]
total.emp.prevhighest <- total.emp$date.label[-1][which(total.emp$value[-1]>=total.emp.current)]
current <- long.last8[long.last8$date==max(long.last8$date) & long.last8$definition=="National",]

clauses <- NULL
for (i in c("increase","decrease","unchanged")){
  if (i=="increase") j <- names(table(current$origin[current$change>0]))
```

```
  if (i=="decrease") j <- names(table(current$origin[current$change<0]))
  if (i=="unchanged") j <- names(table(current$origin[current$change==0]))

  j <- paste(j,collapse =", ")
  j <- stri_replace_last_fixed(j, ',', ' and')
  assign(i,j)

  if (j!=""){
    clauses <- c(clauses,i)
  }

}
sentences <- NULL
for (i in clauses){
  if (i=="increase"){
    j <- increase
    verb <- "risen"
  }
  if (i=="decrease"){
    j <- decrease
    verb <- "fallen"
  }
  if (i=="unchanged"){
    j <- unchanged
    verb <- "is unchanged"
  }

  if (j!=""){
    k <- paste0(verb," for ",j," nationals")
    sentences <- c(sentences,k)
  }
}
sentence <- paste(sentences,collapse="; ")
sentence <- stri_replace_last_fixed(sentence, ';', ' and')
sentence <- gsub(";",",",sentence)
rm(i,j,k,verb,increase,decrease,unchanged,clauses,sentences)
```

Total employment currently stands at 32,641 thousand individuals. Employment has increased year on year by 364 thousand individuals. Total employment is at its highest level since 2018 Q4. Year on year, employment has risen for EU, Non-EU and UK nationals.

Underlying the paragraph is this code:

```
Total employment currently stands at `r format(total.emp.current, scientific=FALSE, big.mark=",")`
thousand individuals. Employment has `r if(total.emp.change>0){"increased"}``r
if(total.emp.change<0){"decreased"}` year on year by `r abs(total.emp.change)` thousand individuals.
Total employment is at its highest level since `r total.emp.prevhighest`. Year on year, employment has
`r sentence`.
```

Concluding Remarks

I have given a brief overview example of producing a report programatically using R Markdown with an API. I used a simple and somewhat messy example. In a real report, you do not have to display all of the code and ouput, as I have here. In other reports, the APIs calls may be much more complex to make, requiring, for instance, prior submission of authenticating information (i.e. username and passwords). Other code may

be more difficult to weave in. For both issues, there is plenty of guidance online (**StackOverflow** is a great resource), but if you have any questions, don't hesitate to contact me (liam.wright.17@ucl.ac.uk). Good luck!