filleR

Filling a (flat) PDF form with data from a CSV file in R

Motivation

As you might know thanks to knitr package it is possible to generate data driven reports in R.

Anyway there are situations when you already have a PDF form intended to be filled and, instead of creating a new filled PDF from scratch, it might be better to use R just to write the filling data over (duplicates of) the existing PDF template.
This strategy is particularly useful when, for legal/administrative reasons, you are required to replicate exactly the original (complex) form.

Note: here by PDF form to be filled I mean a *flat* form, not the special file type hosting form fields.

Requirements

Apart from R, you need Apache PDFBox and a Java runtime environment.

Besides, to position your text properly on the PDF you may take advantage of the *distance* tools present in many PDF applications, including some free ones; unless you may want to print your form and use a ruler or proceed by trial and error.

this image shows a distance tool in action using the free PDF-XChange Viewer.

Among the free viewers for Windows, you might consider PDF-XChange Editor or Foxit Reader.

Usage

Let us assume that `form.pdf` is a one-page A4 PDF representing the template to fill.

Create a `form.csv` file similar to the following:

```
x,y,text,length
3,5,Some text,
3,4,Some text,
1,3,"Very long text to be split every n characters",10
```

```
.....
-1,,,
3,5,Some text,
3,4,Some text,
1,3,"Very long text to be split every n characters",10
.....
```

The first line is just a header, without actual content.
Lines like `x,y,text,` set the x-y coordinates in inches for placing the following `text` on a PDF page based on `form.pdf`, that is, `x`, `y` measure respectively the distance in inches from the left, bottom page border.
Lines like `x,y,text,length` are justified by splitting them in a new line every `length` characters.
`-1„,` stands for a page break, therefore a new page like `form.pdf`is added to the PDF file and the following lines will be printed on it.
Note: CSV outmost commas are not really necessary. Therefore you can write `-1` for `-1„,`.

In order to generate the filled PDF, put in the same directory `form.pdf`, `form.csv`, the filleR.R script, Apache pdfbox-app-x.y.z.jar.

You can now issue in in R:

```
setwd("path/to/script dir")
source("path/to/filleR.R")
makePdf("form.pdf", "form.csv", "filled.pdf")
```

Adjust `setwd("path/to/script dir")` as needed.

If the Java binary executable is in your path, you get the output PDF `filled.pdf`.
Note: to manage executable paths see the specific section ahead.

If your `form.pdf` is not A4, use:

```
makePdf("form.pdf", "form.csv", "filled.pdf", width=X, height=Y)
```

Replace`X` and `Y` with the page width and height in inches.

Finally, you can set a different text magnification level or text font.

```
MAGNI=M
FONT=F
```

Replace `M` with the amount by which text should be magnified relative to the default, current value is `0.7`.

Replace `F` with an integer specifying the font to use. Normally 1 corresponds to plain text (the default), 2 to bold face, 3 to italic and 4 to bold italic. See R `par` function for more insights.

Customise executable paths

filleR finds the Java binary executable first via the R variable

```
JAVABIN="path/to/java-executable"
```

If you don't set it, filleR uses system environment variable `JAVA_HOME`. When set, its value is the path of the directory containing the `bin` directory containing in turn the Java executable.

Besides setting `JAVA_HOME` through the operating system, so that R will inherit it, it is possible to set it in R with:

```
Sys.setenv(JAVA_HOME = "path\\to\\java")
```

This value will last only for the R session.

If `JAVA_HOME` is not set, R will resort to system path. Again it is possible to Java executable directory to the system path environment variable in R:

```
Sys.setenv(PATH = paste(Sys.getenv("PATH"), "path\\to\\java\\binary-dir", sep=.Platform$path
```

Agan this setting is only limted to R session.

As regards the JAR file `pdfbox-app-x.y.z.jar`, it is first found filleR by setting the R variable:

```
PDFBOX="path/to/pdfbox-app-x.y.z.jar"
```

If this variable is not set, filleR looks in the current directory (which might differ from filleR directory) and if still not found it looks into the filleR directory.

How things work internally

R generates the PDF files containing the text, which are like plots with labels only.
Apache pdfbox-app juxtaposes text PDF files over the related PDF templates.

Text PDFs are generated from CSV files which contain the text and the x-y positions, plus page breaks telling to R to generate a new form template.

If you want to learn more about the code read ahead.

Writing on the border of your page

It is easy to create blank plot with `plot.new()` and add text to it with `text()`. The problem is that R adds white spaces every here and there for aesthetic reasons, but, if you need to fill a form, you need to write your text exactly n inch from the borders and not n plus some offset. Obviously you have two alternatives: you adjust your every text printing command to take into account

R blank offsets; you set all R offsets to zero. The former is impractical also because the offset is in percent, so it is not a matter of simple subtracting a given delta. The latter might be a bit tricky, but you do it once and for all.

With the following code you will create `foo.pdf` in R current directory with `hello` written exactly on the left border of the page.

Note that there are three places (in `par()` and `text()`) where we need to nullify the white space.

```
WIDTH=8.3; HEIGHT=11.7        #Paper size A4, measure in inches

pdf('foo.pdf', WIDTH, HEIGHT) #Write next plot to 'foo.pdf'
par(mar=c(0, 0, 0, 0))        #Set numbers of lateral blank lines to zero
par(xaxs='i', yaxs='i')       #Does not extend axes by 4 percent for pretty labels

plot.new()                    #Create a blank plot, where we will want to write our text
plot.window(xlim=c(0,WIDTH), ylim=c(0,HEIGHT)) #Fit plot to paper size
text(0, .5, 'hello', pos=4, offset=0)   #Write without default .5 offset

dev.off()                     #Close device, that is saving for a PDF device
```

Change `WIDTH, HEIGHT` above to your actual paper size.

Since the `text` function will be used quite often and we might want to change font and magnification, it is better to define a specialised print function:

```
###Print left aligned
MAGNI=1    #Magnification factor
FONT=1     #Font: 1 is Helvetica regular, 2 Helv. bold, ... 6 Times
ltext=function(x,y, s) text(x,y, s, pos=4, offset=0, cex=MAGNI, font=FONT)
```

Read the R manual for more information about plot magnification factors and fonts then set them as you please.

Reading data from a CSV file

So we can now easily place text wherever in the page, let's take the data from a CSV file. The structure will be as follows:

```
x,y,text
3,5,John
3,4,Smith
.....
```

As you might have guessed, `x,y` are the coordinates followed by the text to print.

The new code, now reading the overlay material from the CSV data, is:

```
OVER='overlay.pdf'; WIDTH=8.3; HEIGHT=11.7 #Overlay PDF path and size (inches)
```

```
DATA='form.csv'                             #CSV data source

pdf(OVER, WIDTH, HEIGHT) #Write next plot to the overlay PDF
par(mar=c(0, 0, 0, 0))   #Set numbers of lateral blank lines to zero
par(xaxs='i', yaxs='i')  #Does not extend axes by 4 percent for pretty labels

plot.new()                                          #Create the blank plot to write to
plot.window(xlim=c(0,WIDTH), ylim=c(0,HEIGHT)) #Fit plot to paper size
d=read.csv(DATA, as.is=TRUE)                        #Read fill data
ltext(d$x, d$y, d$text)                             #... and print

dev.off()                                           #Save overlay PDF
```

Generating a multipage PDF

The app makes sense if we have multiple forms to fill.

The CSV could now look something like:

```
x,y,text
3,5,John
3,4,Smith
.....
-1
3,5,Bob
3,4,Sullivan
```

`-1` works like a page break and tells to create and skip to a new page.
Note that if you are using a spreadsheet to generate the CSV file, the brek line might look like `-1„`, that will work alike with the following code.

```
OVER='overlay.pdf'; WIDTH=8.3; HEIGHT=11.7 #Overlay PDF path and size (inches)
DATA='form.csv'                             #CSV data source

pdf(OVER, WIDTH, HEIGHT) #Write next plot to the overlay PDF
par(mar=c(0, 0, 0, 0))   #Set numbers of lateral blank lines to zero
par(xaxs='i', yaxs='i')  #Does not extend axes by 4 percent for pretty labels

plot.new()                                          #Create the blank plot to write to
plot.window(xlim=c(0,WIDTH), ylim=c(0,HEIGHT)) #Fit plot to paper size

d=read.csv(DATA, as.is=TRUE)                        #Read and print fill data one row per time
for (i in 1:nrow(d)) {
  x=d[i,1]; y=d[i,2]; tx=d[i,3]
  if(is.na(y)){                                     #On -1 start a new plot/page
    plot.new()
    plot.window(xlim=c(0,WIDTH), ylim=c(0,HEIGHT))
```

```
  }
  ltext(x, y, tx)
}

dev.off()                                          #Save overlay PDF
```

Note that the break works by checking that the second field `y=d[i,2]` in CSV lines is empty.

Adding multiline entries with automatic left justification

Another interesting thing could be to fill a multilne box. The idea is that in the CSV we set an optional `length` field, where we say how many characters the multiline text should be large. So, in the CSV the row for a multiline box would be like:

```
x,y,text,length
3,4,"Very long text to be split every n characters",10
```

Note the double quotes to mask commas, which often recur in long texts.

To left justify a string with a given text width, we define:

```
###Left multiline justification at 'width'
justify=function(string, width){

  str.len=nchar(string)
  sp=gregexpr(' ', string)[[1]]                              #Get text spaces
  l=seq(from=width, by=width, length=floor(str.len/width)) #Get limits for every row
  bsp=sapply(l, function(x) max(sp[sp<=x]))                  #Breaking spaces
  rows=substring (string, c(1, bsp), c(bsp, str.len))        #Extract lines
  rows=sub('^ +', '', rows)                                  #Remove leading spaces
  paste(rows, '\n', collapse='')                             #Merge rows, with newlines
}
```

Integrating the previous code will bring too:

```
OVER='overlay.pdf'; WIDTH=8.3; HEIGHT=11.7 #Overlay PDF path and size (inches)
DATA='form.csv'                            #CSV data source

pdf(OVER, WIDTH, HEIGHT) #Write next plot to the overlay PDF
par(mar=c(0, 0, 0, 0))   #Set numbers of lateral blank lines to zero
par(xaxs='i', yaxs='i')  #Does not extend axes by 4 percent for pretty labels

plot.new()                                         #Create the blank plot to write to
plot.window(xlim=c(0,WIDTH), ylim=c(0,HEIGHT)) #Fit plot to paper size

d=read.csv(DATA, as.is=TRUE)                       #Read and print fill data one row per time
```

```
for (i in 1:nrow(d)) {
  x=d[i,1]; y=d[i,2]; tx=d[i,3]; text.width=d[i,4]

  if(is.na(y)){                                    #On -1 start a new plot/page
    plot.new()
    plot.window(xlim=c(0,WIDTH), ylim=c(0,HEIGHT))
  }

  if(!is.na(text.width)) tx=justify(tx, text.width) #Justify left
  ltext(x, y, tx)
}

dev.off()                                          #Save overlay PDF
```

Overlay the text over the form

To finalize our project we need to print on the form, that is to overlay the generated PDF over the original form.

To make a real world example I will use a book I have to fill for tracking my lectures.

The template for the CSV data to print is as follows:

```
x,y,text,length
*,*,Lecture Day/Month
*,*,Lecture Start time
*,*,Lecture End time
*,*,Lecture Description,15
... 4 like this per page
-1
... start again
```

For a usable 2-page CSV see here.

Here is the form before and after filling (unfortunately not in English).

First of all, to keep track of pages used, we have to add a page counter every time we call `plot.new`. So it could be a good idea to define a new-page function:

```
PAGE.COUNT=0
....
###Create a new overlay page and update the page counter
new.page=function(page.width, page.height){
    plot.new()               #Create a blank plot, as we just want to write our text
    plot.window(xlim=c(0,page.width), ylim=c(0,page.height)) #Fit plot to paper size
    PAGE.COUNT<<-PAGE.COUNT+1
}
```

Here to overlay our text over we will use the open source and cross platform Apache PDFbox and particularly the java based PDFBox Command Line Tools. Before starting make sure that both pdfbox-app and the single page form to be filled (`form.pdf`) are available from the R working path. As an alternative, you may want to adjust the path occurring in the following code in accordance to yours. For portability reason, we will run the PDFBox shell commands via R.

We start initialising PDFBox and related variables:

```
PDFBOX='java -jar pdfbox-app-1.8.2.jar'    #Modify this lines to match your system, version
TEMP='temp.pdf'; FORM='form.pdf'; FILLED='form-filled.pdf'                  #... and your fo
```

We now create a temporary PDF replicating the single-page form for the number of overlay pages using the PDFMerger command, the synopsis is:

```
java -jar pdfbox-app-x.y.z.jar PDFMerger <Source PDF files (2 ..n)> <Target PDF file>
```

So we run:

```
cmd=paste(rep(FORM, PAGE.COUNT), collapse=' ')
cmd=paste(PDFBOX,  'PDFMerger',  cmd, TEMP)
try(system(cmd, intern = TRUE))
```

We can now overlay the overlay PDF on the temporary PDF by means the homonymous command OverlayPDF:

```
java -jar pdfbox-app-x.y.z.jar Overlay <overlay.pdf> <document.pdf> <result.pdf>
```

That is:

```
cmd=paste(PDFBOX, 'OverlayPDF', OVER, TEMP, FILLED)
try(system(cmd, intern = TRUE))
```