An Introduction to the `fifer` Package in R

Dustin A. Fife
Department of Arthritis and Clinical Immunology
Oklahoma Medical Research Foundation

Introduction

The development of this package began in July of 2013. I found myself spending the majority of my time manipulating the dataset and very little of my time actually analyzing the data. As I did, Figure 1 came to mind, and I thought "There's got to be a more efficient way of doing this." Since then I have diligently labored to create an R package for basic data manipulation, as well as preliminary analyses and plotting.

The purpose of this paper is to introduce the `fifer` package and familiarize the reader with the basic functions and how they can be used to simplify data analysis. In the first part, I talk about installing the package. In the second part, I introduce some of the basic data manipulation functions. Next, I show some of the basic functions for data analysis. I end by introducing several plotting functions. Throughout the paper, I try to keep the commentary to a minimum so the user can easily breeze through this without having to digest my witty banter.

Installation

Code

```
### 1. first the package devtools must be installed
install.packages("devtools")
### 2. then we must load the package
require(devtools)
### 3. all that rigamarole to get the function install_github,
### which is how we will install fifer
install_github("fifer", username="dustinfife")
### 4. now load the fifer package
require(fifer)
```

Explanation of Code

Currently, `fifer` is located on github and to install from github requires a special function called `install_github` that is a part of another package `devtools`. The first two steps are simply there to install the `devtools` package so `fifer` can be installed.

Dustin Fife may be contacted at email:fife.dustin@gmail.com

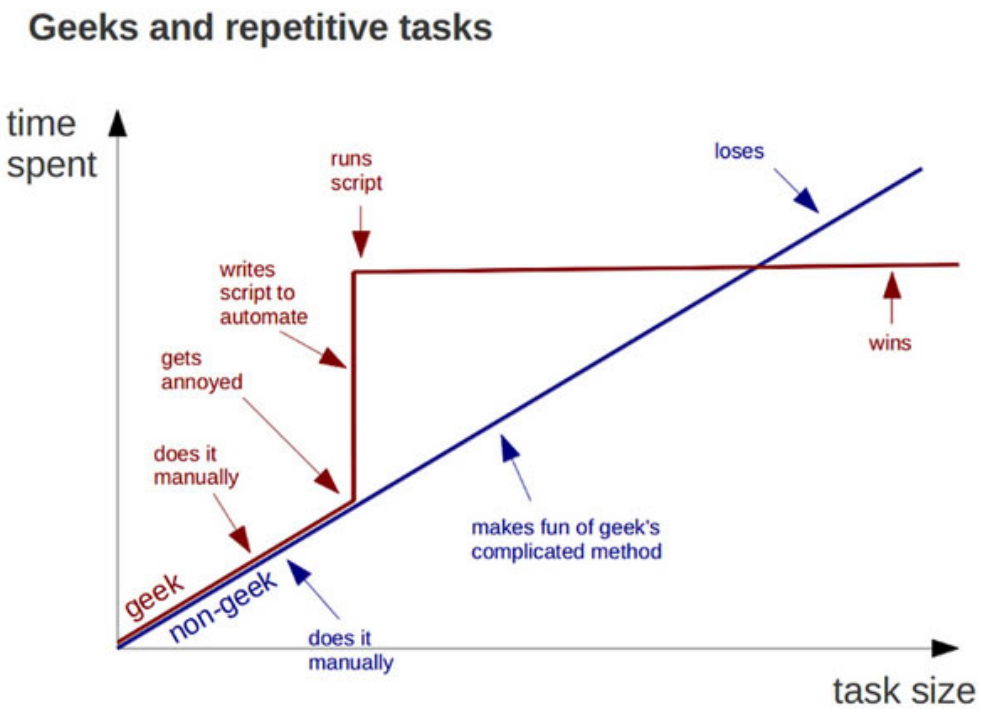

Figure 1. Relationship between time spent and the size of the task for nerds and non-nerds. Pulled from `http://www.globalnerdy.com/2012/04/24/geeks-and-repetitive-tasks/`

Data Manipulation

Introduction

Most of the data manipulation I do involves retrieving an excel file with 16.4×10^{18} columns. In reality, I only need about ten of those columns. In the past, this required opening a massive excel file, waiting, waiting, watching my computer crash, waiting for a restart, opening again, rinse and repeat. When I finally get it open, then I started deleting columns I didn't need until I only had the ten remaining columns.

This method is problematic for two reasons: (1) it is time consuming, and (2) (more importantly) if a change is made at the excel file level, those changes are not reflected in my condensed matrix. With this in mind, I created a series of functions that make it simple to extract only the columns you need.

The `r` Function

Often times, the variables of interest are listed consecutively (e.g., there's a section of demographics that covers 8 columns, there's a section of certain types of biomarkers for 60 columns, then there's a section of clinical information for 18 columns). The `r` function is used to select a consecutive range of columns and requires three arguments: the name of the starting variable, the name of the ending variable, and the names of the dataset. An optional argument tells the computer to return the string names or the column indices.

```
### first load the fakeMedicalData dataset
data(fakeMedicalData)
### show all the column names (well, the first 60 at least)
names(fakeMedicalData)[1:60]
```

```
 [1] "ID"         "disease"    "gender"     "ethnicity"  "age"
 [6] "B_regs_10A" "B_regs_10B" "B_regs_10C" "B_regs_10D" "B_regs_10E"
[11] "B_regs_1A"  "B_regs_1B"  "B_regs_1C"  "B_regs_1D"  "B_regs_1E"
[16] "B_regs_2A"  "B_regs_2B"  "B_regs_2C"  "B_regs_2D"  "B_regs_2E"
[21] "B_regs_3A"  "B_regs_3B"  "B_regs_3C"  "B_regs_3D"  "B_regs_3E"
[26] "B_regs_4A"  "B_regs_4B"  "B_regs_4C"  "B_regs_4D"  "B_regs_4E"
[31] "B_regs_5A"  "B_regs_5B"  "B_regs_5C"  "B_regs_5D"  "B_regs_5E"
[36] "B_regs_6A"  "B_regs_6B"  "B_regs_6C"  "B_regs_6D"  "B_regs_6E"
[41] "B_regs_7A"  "B_regs_7B"  "B_regs_7C"  "B_regs_7D"  "B_regs_7E"
[46] "B_regs_8A"  "B_regs_8B"  "B_regs_8C"  "B_regs_8D"  "B_regs_8E"
[51] "B_regs_9A"  "B_regs_9B"  "B_regs_9C"  "B_regs_9D"  "B_regs_9E"
[56] "BCI_10A"    "BCI_10B"    "BCI_10C"    "BCI_10D"    "BCI_10E"
```

```
### extract all column indices between B_regs_10A and B_regs_9B
bregs = r("B_regs_10A", "B_regs_9E", data.names=names(fakeMedicalData))
bregs
```

```
 [1]  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
[26] 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
```

```
### return the names instead of the column indices
bregs = r("B_regs_10A", "B_regs_9E", data.names=names(fakeMedicalData), names=T)
bregs
```

```
 [1] "B_regs_10A" "B_regs_10B" "B_regs_10C" "B_regs_10D" "B_regs_10E"
 [6] "B_regs_1A"  "B_regs_1B"  "B_regs_1C"  "B_regs_1D"  "B_regs_1E"
[11] "B_regs_2A"  "B_regs_2B"  "B_regs_2C"  "B_regs_2D"  "B_regs_2E"
[16] "B_regs_3A"  "B_regs_3B"  "B_regs_3C"  "B_regs_3D"  "B_regs_3E"
[21] "B_regs_4A"  "B_regs_4B"  "B_regs_4C"  "B_regs_4D"  "B_regs_4E"
[26] "B_regs_5A"  "B_regs_5B"  "B_regs_5C"  "B_regs_5D"  "B_regs_5E"
[31] "B_regs_6A"  "B_regs_6B"  "B_regs_6C"  "B_regs_6D"  "B_regs_6E"
[36] "B_regs_7A"  "B_regs_7B"  "B_regs_7C"  "B_regs_7D"  "B_regs_7E"
[41] "B_regs_8A"  "B_regs_8B"  "B_regs_8C"  "B_regs_8D"  "B_regs_8E"
[46] "B_regs_9A"  "B_regs_9B"  "B_regs_9C"  "B_regs_9D"  "B_regs_9E"
```

But we haven't reached the cool part yet. So far, we have a vector of variable names (or a vector of column indices). What we'd like to do is subset the dataset so that it only gives us the names we want. That brings us to the `make.null` function.

The make.null Function

The `make.null` function takes a series of column names (or indices) and either retains or deletes those columns.

```
### keep only the demographic/b_regs data
newData = make.null("ID","gender", "ethnicity", "age",
                                bregs,
                                data=fakeMedicalData, keep=TRUE)
### or we could drop everything between bregs and the end
newData2 = make.null(
                        r("BCI_10A", "TNF_9E", data.names=names(fakeMedicalData)),
                        data=fakeMedicalData, keep=FALSE)
### check the dimensions of the dataset
dim(fakeMedicalData)
```

```
[1]  60 405
```

```
dim(newData)
```

```
[1] 60 54
```

```
dim(newData2)
```

```
[1] 60 55
```

For more information, type `?make.null` to access the documentation for this function.

The excelMatch Function

Sometimes when people give me data requests, it goes something like this:

> Can you see if disease activity, Column BQ, is related to Blood Pressure (Column MX), Red Blood Cell counts (Column AF), and/or age (Column F)?

The `excelMatch` function allows the user to specify a string (or a vector of strings) corresponding to Excel columns. It will then return the column indices or the actual names of the variables.

```
### extract the variable names corresponding to Excel Columns AA, CD, and FF
excel.names = excelMatch("AA", "CD", "FF", names=names(fakeMedicalData))
excel.names
```

```
[1] "B_regs_4B"  "BCI_5B"     "Glucose_1B"
```

```
### or, we can extract the column indices instead
### (note it does not require names in original dataset)
excel.names = excelMatch("AA", "CD", "FF", n=length(names(fakeMedicalData)))
excel.names
```

```
[1]  27  82 162
```

```
### now subset the matrix to just those using make.null
new.dat = make.null(excel.names, data=fakeMedicalData, keep=T)
head(new.dat)
```

```
  B_regs_4B    BCI_5B Glucose_1B
1  5.438953  7.358441   24.81897
2  7.633085 10.370046   31.46726
3  5.929818  9.916064   22.31351
4  4.703581  7.921184   25.94599
5  5.732308 10.078530   31.70801
6  5.617234  8.341014   27.15707
```

The subsetString function

Often when I import a dataset, the names are just miserable to look at. This is often because the researchers I work with make strange notes to themselves in the columns (e.g., "ANA by IFA 0=neg >40=pos"). R does its best to make sense of it, but it inevitably comes out looking like this: `ANA.by.IFA.0.neg...40.pos`. Often, only the first chunk of information is useful to me (in this case `ANA`). So, I created a function that looks for a separator (in this case a period), then extracts only the first (or only the second, third, etc.) element of a string.

```
#### generate random data (normally this would come from importing a file)
data = data.frame(matrix(rnorm(10*3), ncol=3))
names(data) = c("ANA.by.IFA.0.neg.40...pos",
        "dsDNA....Calculated.",
        "IgG..10.neg..10.19.low..20.89.mod...90.high")
#### print the names (so we can see how messy they are)
names(data)
```

```
[1] "ANA.by.IFA.0.neg.40...pos"
[2] "dsDNA....Calculated."
[3] "IgG..10.neg..10.19.low..20.89.mod...90.high"
```

```
#### rename the column names, taking only the first element
names(data) = subsetString(names(data), sep=".", position=1)
names(data)
```

```
[1] "ANA"   "dsDNA" "IgG"
```

Here, I specified that the separator is a period and that I should take the first element.

I do recommend using caution with this one. Sometimes the naming isn't consistent and applying the same rule across the entire dataset may not work. For example, if the original name was something like "anti-dsDNA, pos>10, neg<10", it would come out as `anti.dsDNA..pos.10..neg.10`, and using the code above would produce `anti`, which isn't what we want.

The write.fife and read.fife functions

Let us suppose that we have used the above functions to create a subsetted dataset (we'll call it `formattedMatrix.csv`). Let us also suppose that some unsavory researcher in our lab decided to update the data matrix and didn't tell us. Unbeknownst to us, our entire analysis is wrong because we are using an outdated matrix. After basking in pride when we see our publication in print, some young arrogant biostatistician accuses you of fabricating your data because he cannot reproduce your results. It isn't until then that you realize with horror the error that you made. After dozens of lawsuits, several public addresses of apology, a half-dozen grant funding removals, and moving to Haiti, you decide something needs to change. So you start using the write.fife and read.fife functions!

What write.fifer does is create a separate file (kinda like meta data) that allows the user to specify the location of the original data file. Then, read.fifer will output that information. This way, the statistician is never too far removed from knowing what the original data file was that created the subsetted matrix.

The example below shows how one might use it.

```
original.path = "documents/research/medical_data_apr_2014.xlsx"
require(xlxx)
d = read.xlsx(original.path, sheetIndex=1, startRow=3)
#### do some data manipulation and create a dataset
#### called d_new (not actually shown)
write.fife(d_new, newfile="documents/research/medicalFormatted.csv",
                 originalfile=original.path, fullpath=T)
```

Now, when we read that file back in, we get the following message:

```
Loading objects:
  original.file
Original File Name: documents/research/medical_data_apr_2014.xlsx
```

Hopefully this will lead to less confusion (and zero lawsuits).

Basic Data Analysis

Hopefully that brief introduction will make data manipulation easier. In this section, I will introduce a series of function that make basic data analysis easier.

The missing.vals Function

My background is in handling missing data, so often the first thing I want to know is what variables have missing information. I created a function called `missing.vals` that does just that. It only requires one argument (a dataset) and it will return a list that indicates which variables have missing values (and how many are missing).

```
missing.vals(fakeMedicalData)
```

```
              Number Missing
B_regs_2C                 18
B_regs_6D                 18
B_regs_8B                 18
BCI_2E                    18
BCI_6A                    18
BCI_6C                    18
Glucose_4E                18
Glucose_7E                18
HemoLeptin_4A             18
HemoLeptin_6C             18
TGF_3C                    18
TGF_5E                    18
TGF_9B                    18
TGF_9D                    18
TNF_1D                    18
B_regs_1B                  6
HemoLeptin_6B              6
TGF_1D                     6
TGF_9C                     6
TNF_10D                    6
```

The demographics Function

Often times, the first step in any paper is to display the demographics. I borrowed a demographics function from the `day2day` package. The user specifies a formula (in this case disease = age + gender + ethnicity) and the function returns the demographics, with disease on the columns and the other variables on the rows. Note the command `latex=FALSE`. When `latex=TRUE`, this function can be easily used to export into a LaTeXdocument for easy table display (see Table 1).

```
demographics(disease~age + gender + ethnicity, data=fakeMedicalData, latex=FALSE)
```

```
                      case          control
age       40.50  sd = 6.96 42.13  sd = 6.64
gender
 Female    15 (50 percent)  11 (37 percent)
 Male      15 (50 percent)  19 (63 percent)
ethnicity
 AA         8 (27 percent)   3 (10 percent)
 EA         7 (23 percent)  11 (37 percent)
 His       12 (40 percent)   8 (27 percent)
 NA         3 (10 percent)   8 (27 percent)
```

Table 1: Demographics of the Fake Medical Dataset

	case (n=30)	control (n=30)
age	40.50 ± 6.96	42.13 ± 6.64
gender		
Female	15 (50%)	11 (37%)
Male	15 (50%)	19 (63%)
ethnicity		
AA	8 (27%)	3 (10%)
EA	7 (23%)	11 (37%)
His	12 (40%)	8 (27%)
NA	3 (10%)	8 (27%)

The make.formula Function

I probably use the `make.formula` function more than anything else. With many analyses, a formula is required to perform the analysis (e.g., `lm(y  x + z)`). Oftentimes, I am doing data mining where the list of variables is quite extensive. Rather than writing a big long formula, I use the `make.formula` function. It requires two strings as arguments: the response variable name and the name of the predictor variable(s). Combining this with the `r` function makes formula specification quite easy.

```
#### list all the variables I want to use using the r function
predictors = r("Glucose_10A", "Glucose_9E", names(fakeMedicalData), names=T)
### make sure it worked!
predictors
```

```
 [1] "Glucose_10A" "Glucose_10B" "Glucose_10C" "Glucose_10D" "Glucose_10E"
 [6] "Glucose_1A"  "Glucose_1B"  "Glucose_1C"  "Glucose_1D"  "Glucose_1E"
[11] "Glucose_2A"  "Glucose_2B"  "Glucose_2C"  "Glucose_2D"  "Glucose_2E"
[16] "Glucose_3A"  "Glucose_3B"  "Glucose_3C"  "Glucose_3D"  "Glucose_3E"
[21] "Glucose_4A"  "Glucose_4B"  "Glucose_4C"  "Glucose_4D"  "Glucose_4E"
[26] "Glucose_5A"  "Glucose_5B"  "Glucose_5C"  "Glucose_5D"  "Glucose_5E"
```

```
[31] "Glucose_6A"  "Glucose_6B"  "Glucose_6C"  "Glucose_6D"  "Glucose_6E"
[36] "Glucose_7A"  "Glucose_7B"  "Glucose_7C"  "Glucose_7D"  "Glucose_7E"
[41] "Glucose_8A"  "Glucose_8B"  "Glucose_8C"  "Glucose_8D"  "Glucose_8E"
[46] "Glucose_9A"  "Glucose_9B"  "Glucose_9C"  "Glucose_9D"  "Glucose_9E"
```

```
### now write the formula
formula = make.formula("disease", predictors)
### and look at it
formula
```

```
disease ~ Glucose_10A + Glucose_10B + Glucose_10C + Glucose_10D +
    Glucose_10E + Glucose_1A + Glucose_1B + Glucose_1C + Glucose_1D +
    Glucose_1E + Glucose_2A + Glucose_2B + Glucose_2C + Glucose_2D +
    Glucose_2E + Glucose_3A + Glucose_3B + Glucose_3C + Glucose_3D +
    Glucose_3E + Glucose_4A + Glucose_4B + Glucose_4C + Glucose_4D +
    Glucose_4E + Glucose_5A + Glucose_5B + Glucose_5C + Glucose_5D +
    Glucose_5E + Glucose_6A + Glucose_6B + Glucose_6C + Glucose_6D +
    Glucose_6E + Glucose_7A + Glucose_7B + Glucose_7C + Glucose_7D +
    Glucose_7E + Glucose_8A + Glucose_8B + Glucose_8C + Glucose_8D +
    Glucose_8E + Glucose_9A + Glucose_9B + Glucose_9C + Glucose_9D +
    Glucose_9E
<environment: 0x7fe63354eb18>
```

The univariate.tests Function

In biostatistics, we often deal with large p/small n datasets (i.e., lots of variables with few people). Often a first filtering step is to perform univariate tests on each of the predictor variables, then narrow down to those that pass statistical significance. The `univariate.tests` function automatically detects which test to use (t-test, ANOVA, or chi-square). See documentation (`?univariate.tests`) for details.

```
#### compute significance tests for each variable in dataset but the ID column
p.values = univariate.tests(dataframe=fakeMedicalData, exclude.cols=1, group="disease")
#### adjust those p-values using FDR (false discovery rate)
p.adjusted = p.adjust(p.values, method="fdr")
#### display only those that exceed statistical significance
p.adjusted[p.adjusted<.05]
```

```
  Glucose_6A HemoLeptin_5A
 0.001515263   0.015493634
```

Plotting

Rather than talking about each plotting function individually, I've included a table (Table 2) that lists many of the plotting functions in the `fifer` package. What follows is sample code showing the many plotting functions.

Table 2: List of functions and their purposes in the `fifer` package.

Function Name	What it does
`auto.layout`	Automatically sets the layout for multiple plots on one page. Good for odd number of plots.
`densityPlotR`	Plot the densities (distributions) of a quantitative variable, conditional on a grouping variable.
`par1`	Automatically sets plotting parameters to my favorite default.
`par2`	Automatically sets plotting parameters to another default.
`prism.plots`	Mimicks the behavior of prism plots where the jittered grouping variable is located on the x-axis and the quantitative variable is on the y-axis, with bars for means or medians
`plotSigBars`	Used in conduction with `prism.plots` to mark which differences are statistically significant.
`string.to.colors`	Given a vector of group labels (e.g., "male", "female", "female", "male", etc.) `string.to.colors` will automatically generate a vector of colors to correspond to the group labels.

```
best.five = names(sort(p.adjusted)[1:5])
### prepare the layout
auto.layout(5)
for (i in 1:length(best.five)){
        ### do my favorite default plotting parameters
        par1()
        ### make a formula
        formula = make.formula(best.five[i], "disease")
        ### plot them
        densityPlotR(formula, data=fakeMedicalData, main="")
}
```

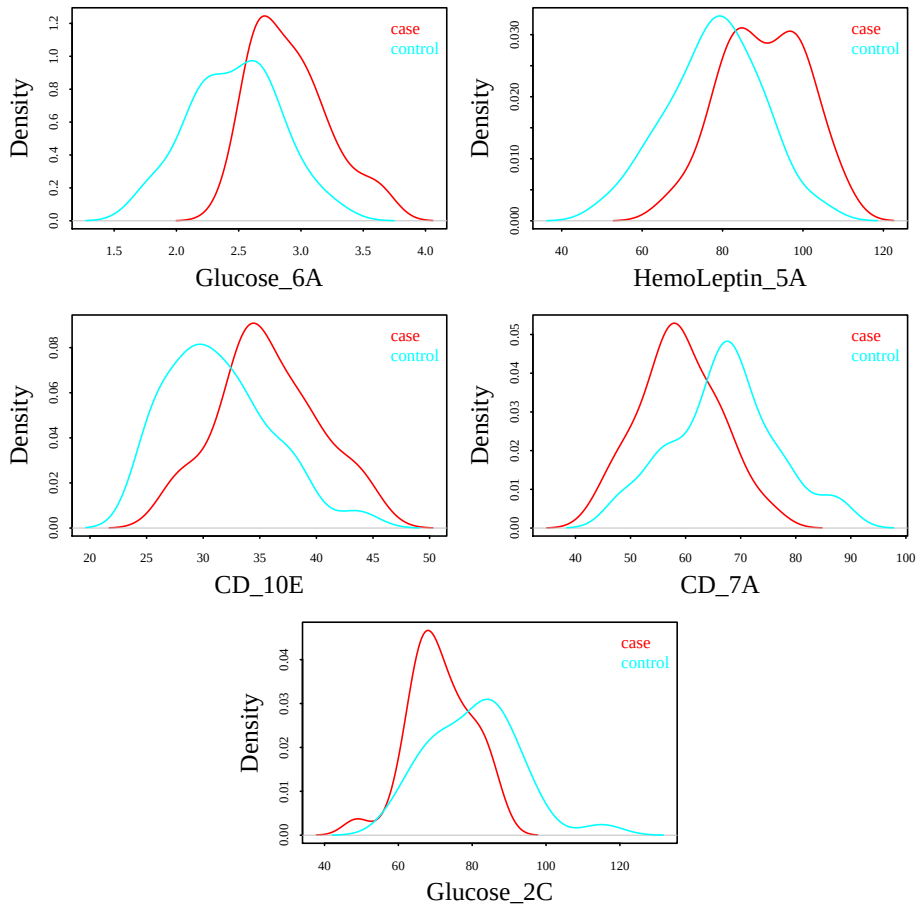

Figure 2. The top five predictors for the fakeMedicalDataset

```
#### set layout again (but only first four)
auto.layout(4)
for (i in 1:4){
        ### do my favorite default plotting parameters
        par1()
        ### make a formula
        formula = make.formula(best.five[i], "disease")
        ### plot them
        prism.plots(formula, data=fakeMedicalData)
        ### show significance bars
        plotSigBars(formula, data=fakeMedicalData, type="tukey")

}
```

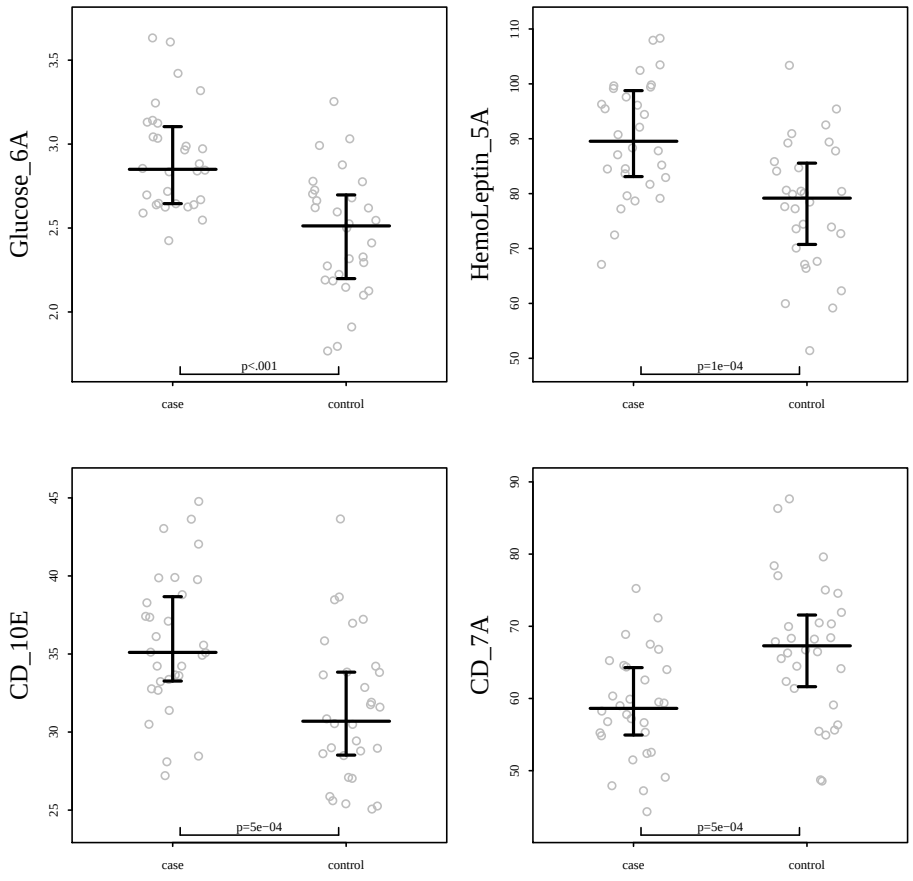

Figure 3. The top four predictors for the fakeMedicalDataset, plotted using densities instead of prism plots

```
        ### change default parameters
par2()
        #### color code according to disease status
colors = string.to.colors(fakeMedicalData$disease, colors=c("blue", "red"))
        #### change symbol according to disease status
pch = as.numeric(string.to.colors(fakeMedicalData$disease, colors=c(15, 16)))
        #### plot it
plot(fakeMedicalData[,best.five[1]], fakeMedicalData[,best.five[2]], col=colors,
                pch=pch, xlab = best.five[1], ylab=best.five[2])
legend("bottomright", c("Case", "Control"), pch=c(15,16),
                col=c("blue", "red"), bty="n")
```

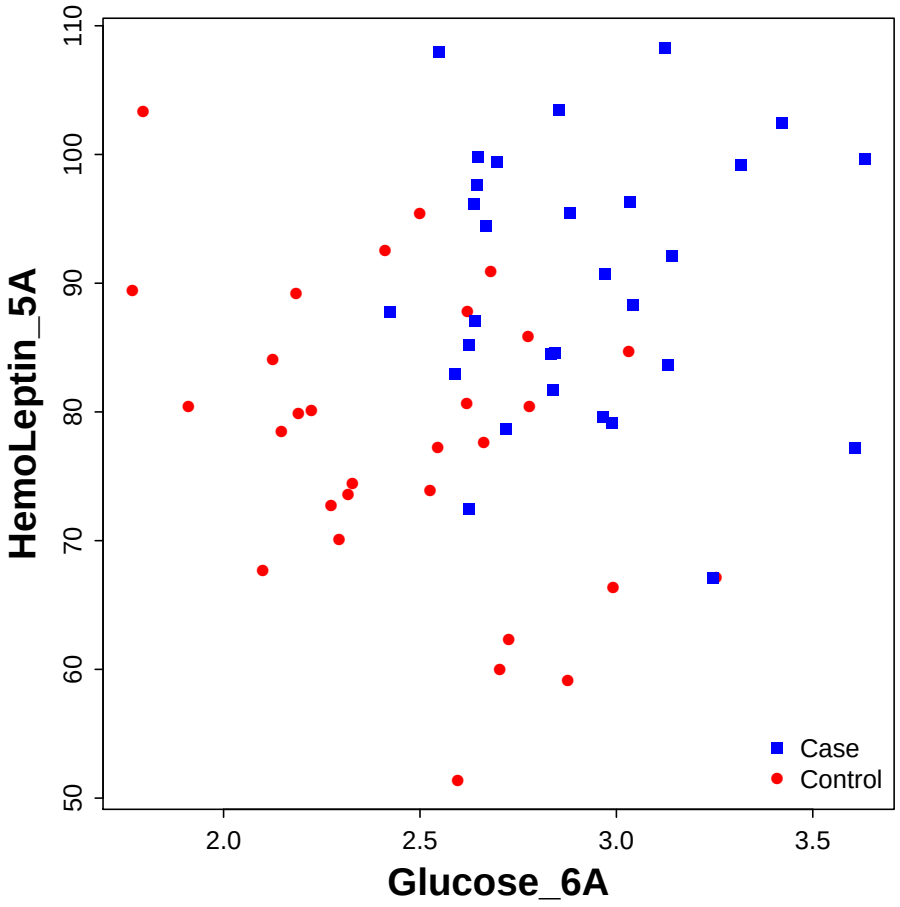

Figure 4. A scatterplot showing that color-codes (and codes with different symbols) the different groups.

```
        #### simulate skewed data (just for the demo)
x = rnorm(100)^2
y = rnorm(100)^2
        ### induce a correlation of .6 (approx) with choselski decomp
cor = matrix(c(1, .6, .6, 1), nrow=2)
skewed.data = cbind(x,y)%*%chol(cor)
names(skewed.data) = c("x", "y")
        #### show original plot
par2()
plot(skewed.data, xlab="x", ylab="y")
```

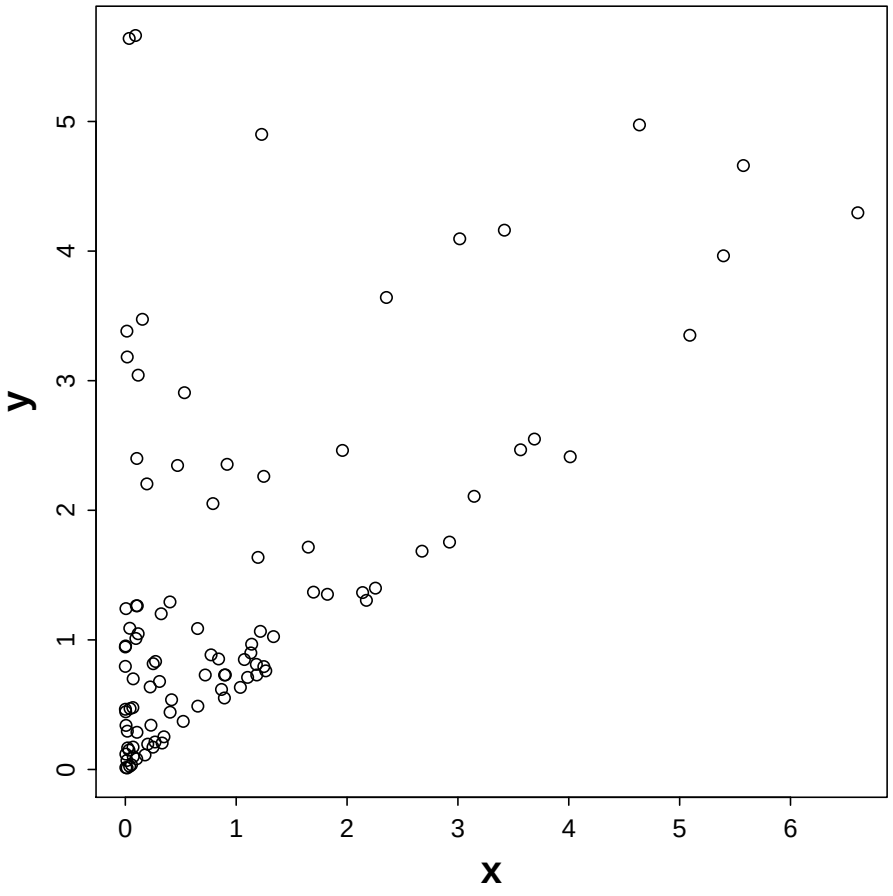

Figure 5. A scatter plot of the skewed data.

```
#### now show the spearman version of the plot
par2()
spearman.plot(skewed.data, xlab="rank(x)", ylab="rank(y)", pch=16)
```

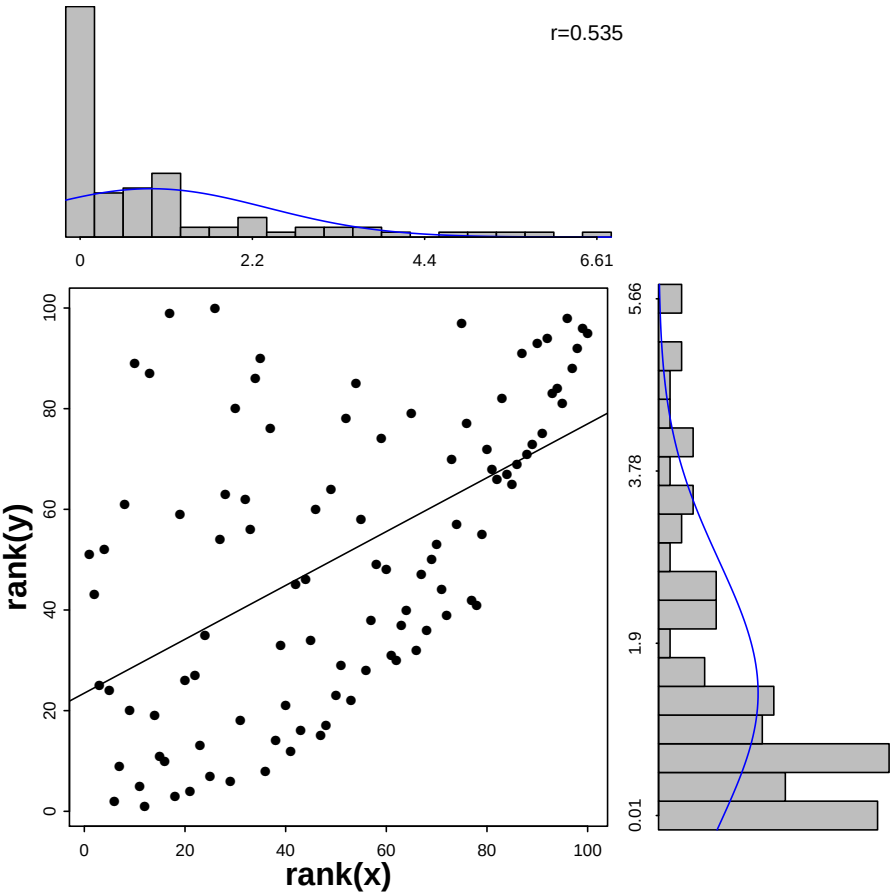

Figure 6. A spearman plot of the skewed data.