The [plumber](https://shirinsplayground.netlify.com/2018/01/plumber/www.rplumber.io) package for R makes it easy to expose existing R code as a webservice via an API (Trestle Technology, LLC 2017).

You take an existing R script and make it accessible with plumber by simply adding a few lines of comments. If you have worked with Roxygen before, e.g. when building a package, you will already be familiar with the core concepts. If not, here are the most important things to know:

* you define the output or endpoint
* you can add additional annotation to customize your input, output and other functionalities of your API
* you can define every input parameter that will go into your function
* every such annotation will begin with either #' or #\*

With this setup, we can take a trained machine learning model and make it available as an API. With this API, other programs can access it and use it to make predictions.

**What are APIs and webservices?**

With plumber, we can build so called **HTTP APIs**. HTTP stands for Hypertext Transfer Protocol and is used to transmit information on the web; API stands for Application Programming Interface and governs the connection between some software and underlying applications. Software can then communicate via HTTP APIs. This way, our R script can be called from other software, even if the other program is not written in R and we have built a tool for machine-to-machine communication, i.e. a webservice.

**How to convert your R script into an API with plumber**

**Training and saving a model**

Let’s say we have trained a machine learning model as in [this post about LIME](https://shirinsplayground.netlify.com/2017/12/lime_sketchnotes/). I loaded a data set on chronic kidney disease, did some preprocessing (converting categorical features into dummy variables, scaling and centering), split it into training and test data and trained a Random Forest model with caret. We can use this trained model to make predictions for one test case with the following code:

library(tidyverse)

# load test and train data

load("../../data/test\_data.RData")

load("../../data/train\_data.RData")

# load model

load("../../data/model\_rf.RData")

# take first test case for prediction

input\_data <- test\_data[1, ] %>%

select(-class)

# predict test case using model

pred <- predict(model\_rf, input\_data)

cat("----------------\nTest case predicted to be", as.character(pred), "\n----------------")

## ----------------

## Test case predicted to be ckd

## ----------------

**The input**

For our API to work, we need to define the input, in our case the features of the test data. When we look at the model object, we see that it expects the following parameters:

var\_names <- model\_rf$finalModel$xNames

var\_names

## [1] "age" "bp" "sg\_1.005" "sg\_1.010"

## [5] "sg\_1.015" "sg\_1.020" "sg\_1.025" "al\_0"

## [9] "al\_1" "al\_2" "al\_3" "al\_4"

## [13] "al\_5" "su\_0" "su\_1" "su\_2"

## [17] "su\_3" "su\_4" "su\_5" "rbc\_normal"

## [21] "rbc\_abnormal" "pc\_normal" "pc\_abnormal" "pcc\_present"

## [25] "pcc\_notpresent" "ba\_present" "ba\_notpresent" "bgr"

## [29] "bu" "sc" "sod" "pot"

## [33] "hemo" "pcv" "wbcc" "rbcc"

## [37] "htn\_yes" "htn\_no" "dm\_yes" "dm\_no"

## [41] "cad\_yes" "cad\_no" "appet\_good" "appet\_poor"

## [45] "pe\_yes" "pe\_no" "ane\_yes" "ane\_no"

Good practice is to write the input parameter definition into you [API Swagger UI](https://swagger.io/swagger-ui/), but the code would work without these annotations. We define the parameters by annotating them with name and description in our R-script using @parameter. For this purpose, I want to know the type and min/max values for each of my variables in the training data. Because categorical data has been converted to dummy variables and then scaled and centered, these values will all be numeric and between 0 and 1 in this example. If I would build this script for a real case, I’d use the raw data as input and add a preprocessing function to my script, though!

# show parameter definition for the first three features

for (i in 1:3) {

# if you wanted to see it for all features, use

#for (i in 1:length(var\_names)) {

var <- var\_names[i]

train\_data\_subs <- train\_data[, which(colnames(train\_data) == var)]

type <- class(train\_data\_subs)

if (type == "numeric") {

min <- min(train\_data\_subs)

max <- max(train\_data\_subs)

}

cat("Variable:", var, "is of type:", type, "\n",

"Min value in training data =", min, "\n",

"Max value in training data =", max, "\n----------\n")

}

## Variable: age is of type: numeric

## Min value in training data = 0

## Max value in training data = 0.9777778

## ----------

## Variable: bp is of type: numeric

## Min value in training data = 0

## Max value in training data = 0.7222222

## ----------

## Variable: sg\_1.005 is of type: numeric

## Min value in training data = 0

## Max value in training data = 1

## ----------

Unless otherwise instructed, all parameters passed into plumber endpoints from query strings or dynamic paths will be character strings. <https://www.rplumber.io/docs/routing-and-input.html#typed-dynamic-routes>

This means that we need to convert numeric values before we process them further. Or we can define the parameter type explicitly, e.g. by writing variable\_1:numeric if we want to specifiy that *variable\_1* is supposed to be numeric.

To make sure that the model will perform as expected, it is also advisable to add a few validation functions. Here, I will validate

* whether every parameter is numeric/integer by checking for NAs (which would have resulted from as.numeric()/as.integer() applied to data of character type)
* whether every parameter is between 0 and 1

In order for plumber to work with our input, it needs to be part of the HTTP request, which can then be routed to our R function. The [plumber documentation](https://www.rplumber.io/docs/routing-and-input.html#query-strings) describes how to use query strings as inputs. But in our case, manually writing query strings is not practical because we have so many parameters. Of course, there are programs that let us generate query strings but the easiest way to format the input from a line of data I found is to use JSON.

The toJSON() function from the rjson package converts our input line to JSON format:

library(rjson)

test\_case\_json <- toJSON(input\_data)

cat(test\_case\_json)

## {"age":0.511111111111111,"bp":0.111111111111111,"sg\_1.005":1,"sg\_1.010":0,"sg\_1.015":0,"sg\_1.020":0,"sg\_1.025":0,"al\_0":0,"al\_1":0,"al\_2":0,"al\_3":0,"al\_4":1,"al\_5":0,"su\_0":1,"su\_1":0,"su\_2":0,"su\_3":0,"su\_4":0,"su\_5":0,"rbc\_normal":1,"rbc\_abnormal":0,"pc\_normal":0,"pc\_abnormal":1,"pcc\_present":1,"pcc\_notpresent":0,"ba\_present":0,"ba\_notpresent":1,"bgr":0.193877551020408,"bu":0.139386189258312,"sc":0.0447368421052632,"sod":0.653374233128834,"pot":0,"hemo":0.455056179775281,"pcv":0.425925925925926,"wbcc":0.170454545454545,"rbcc":0.225,"htn\_yes":1,"htn\_no":0,"dm\_yes":0,"dm\_no":1,"cad\_yes":0,"cad\_no":1,"appet\_good":0,"appet\_poor":1,"pe\_yes":1,"pe\_no":0,"ane\_yes":1,"ane\_no":0}

**Defining the endpoint and output**

In order to convert this very simple script into an API, we need to define the endpoint(s). Endpoints will return an output, in our case it will return the output of the predict() function pasted into a line of text (e.g. “Test case predicted to be ckd”). Here, we want to have the predictions returned, so we annotate the entire function with @get. This endpoint in the API will get a custom name, so that we can call it later; here we call it predict and therefore write #' @get /predict.

According to the design of the HTTP specification, GET (along with HEAD) requests are used only to read data and not change it. Therefore, when used this way, they are considered safe. That is, they can be called without risk of data modification or corruption — calling it once has the same effect as calling it 10 times, or none at all. Additionally, GET (and HEAD) is idempotent, which means that making multiple identical requests ends up having the same result as a single request. <http://www.restapitutorial.com/lessons/httpmethods.html>

In this case, we could also consider using @post to avoid caching issues, but for this example I’ll leave it as @get.

The POST verb is most-often utilized to **create** new resources. In particular, it’s used to create subordinate resources. That is, subordinate to some other (e.g. parent) resource. In other words, when creating a new resource, POST to the parent and the service takes care of associating the new resource with the parent, assigning an ID (new resource URI), etc. On successful creation, return HTTP status 201, returning a Location header with a link to the newly-created resource with the 201 HTTP status. POST is neither safe nor idempotent. It is therefore recommended for non-idempotent resource requests. Making two identical POST requests will most-likely result in two resources containing the same information. <http://www.restapitutorial.com/lessons/httpmethods.html>

We can also customize the output. Keep in mind though, that the output should be [“serialized”](https://www.rplumber.io/docs/rendering-and-output.html#serializers). By default, the output will be in JSON format. Here, I want to have a text output, so I’ll specify @html without html formatting specifications, although I could add them if I wanted to display the text on a website. If we were to [store the data in a database](https://www.rplumber.io/docs/runtime.html#external-data-store), however, this would not be a good idea. In that case, it would be better to output the result as a JSON object.

**Logging with filters**

It is also useful to provide some sort of logging for your API. Here, I am using the simple example from the [plumber documentation](https://www.rplumber.io/docs/routing-and-input.html#filters) that uses filters and output the logs to the console or your API server logs. You could also [write your logging output to a file](https://www.rplumber.io/docs/runtime.html#file-system). In production, it would be better to use a real logging setup that stores information about each request, e.g. the time stamp, whether any errors or warnings occurred, etc. The forward() part of the logging function passes control on to the next handler in the pipeline, here our predict function.

**Running the plumber script**

We need to save the entire script with annotations as an *.R* file as seen below. The regular comments # describe what each section does.

# script name:

# plumber.R

# set API title and description to show up in http://localhost:8000/\_\_swagger\_\_/

#' @apiTitle Run predictions for Chronic Kidney Disease with Random Forest Model

#' @apiDescription This API takes as patient data on Chronic Kidney Disease and returns a prediction whether the lab values

#' indicate Chronic Kidney Disease (ckd) or not (notckd).

#' For details on how the model is built, see https://shirinsplayground.netlify.com/2017/12/lime\_sketchnotes/

#' For further explanations of this plumber function, see https://shirinsplayground.netlify.com/2018/01/plumber/

# load model

# this path would have to be adapted if you would deploy this

load("/Users/shiringlander/Documents/Github/shirinsplayground/data/model\_rf.RData")

#' Log system time, request method and HTTP user agent of the incoming request

#' @filter logger

function(req){

cat("System time:", as.character(Sys.time()), "\n",

"Request method:", req$REQUEST\_METHOD, req$PATH\_INFO, "\n",

"HTTP user agent:", req$HTTP\_USER\_AGENT, "@", req$REMOTE\_ADDR, "\n")

plumber::forward()

}

# core function follows below:

# define parameters with type and description

# name endpoint

# return output as html/text

# specify 200 (okay) return

#' predict Chronic Kidney Disease of test case with Random Forest model

#' @param age:numeric The age of the patient, numeric (scaled and centered to be btw 0 and 1)

#' @param bp:numeric The blood pressure of the patient, numeric (scaled and centered to be btw 0 and 1)

#' @param sg\_1.005:int The urinary specific gravity of the patient, integer (1: sg = 1.005, otherwise 0)

#' @param sg\_1.010:int The urinary specific gravity of the patient, integer (1: sg = 1.010, otherwise 0)

#' @param sg\_1.015:int The urinary specific gravity of the patient, integer (1: sg = 1.015, otherwise 0)

#' @param sg\_1.020:int The urinary specific gravity of the patient, integer (1: sg = 1.020, otherwise 0)

#' @param sg\_1.025:int The urinary specific gravity of the patient, integer (1: sg = 1.025, otherwise 0)

#' @param al\_0:int The urine albumin level of the patient, integer (1: al = 0, otherwise 0)

#' @param al\_1:int The urine albumin level of the patient, integer (1: al = 1, otherwise 0)

#' @param al\_2:int The urine albumin level of the patient, integer (1: al = 2, otherwise 0)

#' @param al\_3:int The urine albumin level of the patient, integer (1: al = 3, otherwise 0)

#' @param al\_4:int The urine albumin level of the patient, integer (1: al = 4, otherwise 0)

#' @param al\_5:int The urine albumin level of the patient, integer (1: al = 5, otherwise 0)

#' @param su\_0:int The sugar level of the patient, integer (1: su = 0, otherwise 0)

#' @param su\_1:int The sugar level of the patient, integer (1: su = 1, otherwise 0)

#' @param su\_2:int The sugar level of the patient, integer (1: su = 2, otherwise 0)

#' @param su\_3:int The sugar level of the patient, integer (1: su = 3, otherwise 0)

#' @param su\_4:int The sugar level of the patient, integer (1: su = 4, otherwise 0)

#' @param su\_5:int The sugar level of the patient, integer (1: su = 5, otherwise 0)

#' @param rbc\_normal:int The red blood cell count of the patient, integer (1: rbc = normal, otherwise 0)

#' @param rbc\_abnormal:int The red blood cell count of the patient, integer (1: rbc = abnormal, otherwise 0)

#' @param pc\_normal:int The pus cell level of the patient, integer (1: pc = normal, otherwise 0)

#' @param pc\_abnormal:int The pus cell level of the patient, integer (1: pc = abnormal, otherwise 0)

#' @param pcc\_present:int The puc cell clumps status of the patient, integer (1: pcc = present, otherwise 0)

#' @param pcc\_notpresent:int The puc cell clumps status of the patient, integer (1: pcc = notpresent, otherwise 0)

#' @param ba\_present:int The bacteria status of the patient, integer (1: ba = present, otherwise 0)

#' @param ba\_notpresent:int The bacteria status of the patient, integer (1: ba = notpresent, otherwise 0)

#' @param bgr:numeric The blood glucose random level of the patient, numeric (scaled and centered to be btw 0 and 1)

#' @param bu:numeric The blood urea level of the patient, numeric (scaled and centered to be btw 0 and 1)

#' @param sc:numeric The serum creatinine level of the patient, numeric (scaled and centered to be btw 0 and 1)

#' @param sod:numeric The sodium level of the patient, numeric (scaled and centered to be btw 0 and 1)

#' @param pot:numeric The potassium level of the patient, numeric (scaled and centered to be btw 0 and 1)

#' @param hemo:numeric The hemoglobin level of the patient, numeric (scaled and centered to be btw 0 and 1)

#' @param pcv:numeric The packed cell volume of the patient, numeric (scaled and centered to be btw 0 and 1)

#' @param wbcc:numeric The white blood cell count of the patient, numeric (scaled and centered to be btw 0 and 1)

#' @param rbcc:numeric The red blood cell count of the patient, numeric (scaled and centered to be btw 0 and 1)

#' @param htn\_yes:int The hypertension status of the patient, integer (1: htn = yes, otherwise 0)

#' @param htn\_no:int The hypertension status of the patient, integer (1: htn = no, otherwise 0)

#' @param dm\_yes:int The diabetes mellitus status of the patient, integer (1: dm = yes, otherwise 0)

#' @param dm\_no:int The diabetes mellitus status of the patient, integer (1: dm = no, otherwise 0)

#' @param cad\_yes:int The coronary artery disease status of the patient, integer (1: cad = yes, otherwise 0)

#' @param cad\_no:int The coronary artery disease status of the patient, integer (1: cad = no, otherwise 0)

#' @param appet\_good:int The appetite of the patient, integer (1: appet = good, otherwise 0)

#' @param appet\_poor:int The appetite of the patient, integer (1: appet = poor, otherwise 0)

#' @param pe\_yes:int The pedal edema status of the patient, integer (1: pe = yes, otherwise 0)

#' @param pe\_no:int The pedal edema status of the patient, integer (1: pe = no, otherwise 0)

#' @param ane\_yes:int The anemia status of the patient, integer (1: ane = yes, otherwise 0)

#' @param ane\_no:int The anemia status of the patient, integer (1: ane = no, otherwise 0)

#' @get /predict

#' @html

#' @response 200 Returns the class (ckd or notckd) prediction from the Random Forest model; ckd = Chronic Kidney Disease

calculate\_prediction <- function(age, bp, sg\_1.005, sg\_1.010, sg\_1.015, sg\_1.020, sg\_1.025, al\_0, al\_1, al\_2,

al\_3, al\_4, al\_5, su\_0, su\_1, su\_2, su\_3, su\_4, su\_5, rbc\_normal, rbc\_abnormal, pc\_normal, pc\_abnormal,

pcc\_present, pcc\_notpresent, ba\_present, ba\_notpresent, bgr, bu, sc, sod, pot, hemo, pcv,

wbcc, rbcc, htn\_yes, htn\_no, dm\_yes, dm\_no, cad\_yes, cad\_no, appet\_good, appet\_poor, pe\_yes, pe\_no,

ane\_yes, ane\_no) {

# make data frame from numeric parameters

input\_data\_num <<- data.frame(age, bp, bgr, bu, sc, sod, pot, hemo, pcv, wbcc, rbcc,

stringsAsFactors = FALSE)

# and make sure they really are numeric

input\_data\_num <<- as.data.frame(t(sapply(input\_data\_num, as.numeric)))

# make data frame from (binary) integer parameters

input\_data\_int <<- data.frame(sg\_1.005, sg\_1.010, sg\_1.015, sg\_1.020, sg\_1.025, al\_0, al\_1, al\_2,

al\_3, al\_4, al\_5, su\_0, su\_1, su\_2, su\_3, su\_4, su\_5, rbc\_normal, rbc\_abnormal, pc\_normal, pc\_abnormal,

pcc\_present, pcc\_notpresent, ba\_present, ba\_notpresent, htn\_yes, htn\_no, dm\_yes, dm\_no,

cad\_yes, cad\_no, appet\_good, appet\_poor, pe\_yes, pe\_no, ane\_yes, ane\_no,

stringsAsFactors = FALSE)

# and make sure they really are numeric

input\_data\_int <<- as.data.frame(t(sapply(input\_data\_int, as.integer)))

# combine into one data frame

input\_data <<- as.data.frame(cbind(input\_data\_num, input\_data\_int))

# validation for parameter

if (any(is.na(input\_data))) {

res$status <- 400

res$body <- "Parameters have to be numeric or integers"

}

if (any(input\_data < 0) || any(input\_data > 1)) {

res$status <- 400

res$body <- "Parameters have to be between 0 and 1"

}

# predict and return result

pred\_rf <<- predict(model\_rf, input\_data)

paste("----------------\nTest case predicted to be", as.character(pred\_rf), "\n----------------\n")

}

Note that I am using the “double-assignment” operator <<- in my function, because I want to make sure that objects are overwritten at the top level (i.e. globally). This would have been relevant had I set a global parameter, but to show it the example, I decided to use it here as well.

We can now call our script with the plumb() function, run it with run() and open it on port 800. Calling plumb() creates an environment in which all our functions are evaluated.

library(plumber)

r <- plumb("/Users/shiringlander/Documents/Github/shirinsplayground/static/scripts/plumber.R")

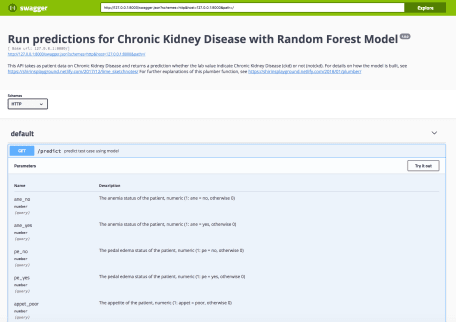
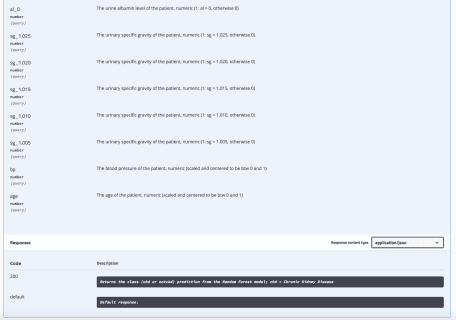
r$run(port = 8000)

We will now see the following message in our R console:

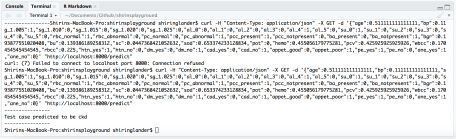
Starting server to listen on port 8000

Running the swagger UI at http://127.0.0.1:8000/\_\_swagger\_\_/

If you go to \*[http://localhost:8000/\_\_swagger\_\_/\*](http://localhost:8000/__swagger__/*), you could now try out the function by manually choosing values for all the parameters we defined in the script.

… 

Because we annotated the calculate\_prediction() function in our script with #' @get /predict we can access it via \*[http://localhost:8000/predict\*](http://localhost:8000/predict*). But because we have no input specified as of yet, we will only see an error on this site. So, we still need to put our JSON formatted input into the function. To do this, we can use [*curl*](https://en.wikipedia.org/wiki/CURL) from the terminal and feed in the JSON string from above. If you are using RStudio in the latest version, you have a handy terminal window open in your working directory. You find it right next to the Console.



Terminal in RStudio

curl -H "Content-Type: application/json" -X GET -d '{"age":0.511111111111111,"bp":0.111111111111111,"sg\_1.005":1,"sg\_1.010":0,"sg\_1.015":0,"sg\_1.020":0,"sg\_1.025":0,"al\_0":0,"al\_1":0,"al\_2":0,"al\_3":0,"al\_4":1,"al\_5":0,"su\_0":1,"su\_1":0,"su\_2":0,"su\_3":0,"su\_4":0,"su\_5":0,"rbc\_normal":1,"rbc\_abnormal":0,"pc\_normal":0,"pc\_abnormal":1,"pcc\_present":1,"pcc\_notpresent":0,"ba\_present":0,"ba\_notpresent":1,"bgr":0.193877551020408,"bu":0.139386189258312,"sc":0.0447368421052632,"sod":0.653374233128834,"pot":0,"hemo":0.455056179775281,"pcv":0.425925925925926,"wbcc":0.170454545454545,"rbcc":0.225,"htn\_yes":1,"htn\_no":0,"dm\_yes":0,"dm\_no":1,"cad\_yes":0,"cad\_no":1,"appet\_good":0,"appet\_poor":1,"pe\_yes":1,"pe\_no":0,"ane\_yes":1,"ane\_no":0}' "http://localhost:8000/predict"

**-H** defines an extra header to include in the request when sending HTTP to a server (<https://curl.haxx.se/docs/manpage.html#-H>).

**-X** pecifies a custom request method to use when communicating with the HTTP server (<https://curl.haxx.se/docs/manpage.html#-X>).

**-d** sends the specified data in a request to the HTTP server, in the same way that a browser does when a user has filled in an HTML form and presses the submit button. This will cause curl to pass the data to the server using the content-type application/x-www-form-urlencoded (<https://curl.haxx.se/docs/manpage.html#-d>).

This will return the following output:

* cat() outputs to the R console if you use R interactively; if you use R on a server, it will be included in the server logs.

System time: 2018-01-15 13:34:32

Request method: GET /predict

HTTP user agent: curl/7.54.0 @ 127.0.0.1

* paste outputs to the terminal

----------------

Test case predicted to be ckd

----------------

**Security**

This example shows a pretty simply R-script API. But if you plan on deploying your API to production, you should consider the [security section of the plumber documentation](https://www.rplumber.io/docs/security.html). It give additional information about how you can make your code (more) secure.

**Finalize**

If you wanted to deploy this API you would need to [host](https://www.rplumber.io/docs/hosting.html) it, i.e. provide the model and run an R environment with plumber, ideally on a server. A good way to do this, would be to package everything in a [Docker](https://www.rplumber.io/docs/hosting.html#docker) container and run this. Docker will ensure that you have a working snapshot of the system settings, R and package versions that won’t change. For more information on dockerizing your API, check out <https://hub.docker.com/r/trestletech/plumber/>.

sessionInfo()

## R version 3.4.2 (2017-09-28)

## Platform: x86\_64-apple-darwin15.6.0 (64-bit)

## Running under: macOS High Sierra 10.13.2

##

## Matrix products: default

## BLAS: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRblas.0.dylib

## LAPACK: /Library/Frameworks/R.framework/Versions/3.4/Resources/lib/libRlapack.dylib

##

## locale:

## [1] de\_DE.UTF-8/de\_DE.UTF-8/de\_DE.UTF-8/C/de\_DE.UTF-8/de\_DE.UTF-8

##

## attached base packages:

## [1] methods stats graphics grDevices utils datasets base

##

## other attached packages:

## [1] plumber\_0.4.4 rjson\_0.2.15 forcats\_0.2.0 stringr\_1.2.0

## [5] dplyr\_0.7.4 purrr\_0.2.4 readr\_1.1.1 tidyr\_0.7.2

## [9] tibble\_1.3.4 ggplot2\_2.2.1 tidyverse\_1.2.1

##

## loaded via a namespace (and not attached):

## [1] httr\_1.3.1 ddalpha\_1.3.1 sfsmisc\_1.1-1

## [4] jsonlite\_1.5 splines\_3.4.2 foreach\_1.4.3

## [7] prodlim\_1.6.1 modelr\_0.1.1 assertthat\_0.2.0

## [10] stats4\_3.4.2 DRR\_0.0.2 cellranger\_1.1.0

## [13] yaml\_2.1.15 robustbase\_0.92-8 ipred\_0.9-6

## [16] backports\_1.1.1 lattice\_0.20-35 glue\_1.2.0

## [19] digest\_0.6.12 randomForest\_4.6-12 rvest\_0.3.2

## [22] colorspace\_1.3-2 recipes\_0.1.1 httpuv\_1.3.5

## [25] htmltools\_0.3.6 Matrix\_1.2-12 plyr\_1.8.4

## [28] psych\_1.7.8 timeDate\_3042.101 pkgconfig\_2.0.1

## [31] CVST\_0.2-1 broom\_0.4.3 haven\_1.1.0

## [34] caret\_6.0-77 bookdown\_0.5 scales\_0.5.0

## [37] gower\_0.1.2 lava\_1.5.1 withr\_2.1.0

## [40] nnet\_7.3-12 lazyeval\_0.2.1 cli\_1.0.0

## [43] mnormt\_1.5-5 survival\_2.41-3 magrittr\_1.5

## [46] crayon\_1.3.4 readxl\_1.0.0 evaluate\_0.10.1

## [49] nlme\_3.1-131 MASS\_7.3-47 xml2\_1.1.1

## [52] dimRed\_0.1.0 foreign\_0.8-69 class\_7.3-14

## [55] blogdown\_0.3 tools\_3.4.2 hms\_0.4.0

## [58] kernlab\_0.9-25 munsell\_0.4.3 bindrcpp\_0.2

## [61] compiler\_3.4.2 RcppRoll\_0.2.2 rlang\_0.1.4

## [64] grid\_3.4.2 iterators\_1.0.8 rstudioapi\_0.7

## [67] rmarkdown\_1.8 gtable\_0.2.0 ModelMetrics\_1.1.0

## [70] codetools\_0.2-15 reshape2\_1.4.2 R6\_2.2.2

## [73] lubridate\_1.7.1 knitr\_1.17 bindr\_0.1

## [76] rprojroot\_1.2 stringi\_1.1.6 parallel\_3.4.2

## [79] Rcpp\_0.12.14 rpart\_4.1-11 tidyselect\_0.2.3

## [82] DEoptimR\_1.0-8