getWBData tutorial

Overview

The code{getWBData} package is a set of functions meant to make downloading and processing data from the westbrook database faster and easier. The functions are not comprehensive; you will still need to manipulate data for your particular application. The most complete data pipeline that the functions facilitate is the downloading and formatting of data to run CJS models in jags using electrofishing data. If the functions are not behaving as you wish, there is also the option of just downloading the data tables directly from the database and doing the data manipulation and joining yourself.

Connecting to the database

Load the package

```
library(getWBData)
```

```
## Loading required package: dplyr

## Warning: package 'dplyr' was built under R version 3.2.3

##
## Attaching package: 'dplyr'
##
## The following objects are masked from 'package:stats':
##
##     filter, lag
##
## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
##
## Loading required package: RPostgreSQL
## Loading required package: DBI
```

To create a connection to the database:

```
reconnect()
```

This will ask for your postgres username and password, and it will clear the console and create two objects that are connections to the database. If you use the built-in functions to get data, you won't need reconnect() or the objects it creates directly, but they will be called internally. However, understanding how to view and download data from the database is certain to be useful for understanding what data are availabe and what the tables and columns are named called. If you ever get an error relating to a postgres connections, rerunning reconnect() should fix it. Sourcing an entire script will not cause problems because the code will pause while you put in credentials, but it you source a chunk (i.e., using ctr + enter) when there is no connection it will enter subsequent lines as the username and password and cause an error. Manually reconnecting before running a chunk will avoid this issue.

conDplyr is a connection made using the Dplyr package, which can be used as follows:

```
#view the tables in the database
conDplyr
```

```
## src:  postgres 9.3.11 [evan@osensei.cns.umass.edu:5432/westbrook]
## tbls: antenna_deployment, data_acoustic, data_boundary_detections,
##   data_by_tag, data_daily_discharge, data_daily_temperature, data_family,
##   data_flow_extension, data_hourly_temperature, data_portable_antenna,
##   data_season, data_seasonal_sampling, data_smolts,
##   data_stationary_antenna, data_tagged_captures, data_trap_captures,
##   data_untagged_captures, data_yoy_bins, family, import_dates,
##   last_data_import, last_imports, raw_captures, raw_depth, raw_family,
##   raw_stanley_acoustic_data, raw_stanley_antenna, raw_stanley_cohorts,
##   raw_stanley_dead_tags, raw_stanley_fyke_net, raw_stanley_tags,
##   raw_tags_antenna, raw_tags_dead, raw_tags_salmon_wb, raw_tags_tribs_wb,
##   raw_tags_trout_wb, raw_temps, raw_yoy_bins, season_map,
##   stanley_environmental, tags_acoustic, tags_antenna,
##   tags_antenna_2011_2015, tags_captures, tags_dead, untagged_captures,
##   yoy_bins
```

```
#connect to a table and then download it
data<-tbl(conDplyr,"data_tagged_captures") %>% #dplyr piping sends the output of this to the first argu
        filter(river=="wb jimmy") %>% #option to filter to the rows desired
        select(detection_date,tag) %>% #or select particular columns
        collect(n=Inf) #download the data into R and closes the connection
print(data)
```

```
## Source: local data frame [5,760 x 2]
##
##    detection_date        tag
##            (time)      (chr)
## 1      2014-09-15 00088cbeff
## 2      2012-09-17 00088cbf1b
## 3      2013-03-26 00088cbf26
## 4      2012-06-04 00088cbf43
## 5      2012-09-17 00088cbf43
## 6      2014-09-15 00088cbf50
## 7      2013-03-26 00088cbf70
## 8      2014-09-15 00088cbf72
## 9      2015-06-08 00088cbf72
## 10     2014-09-15 00088cbf8f
## ..            ...        ...
```

```
#figure out what columns are in a table
tbl(conDplyr,"data_tagged_captures")
```

```
## Source: postgres 9.3.11 [evan@osensei.cns.umass.edu:5432/westbrook]
## From: data_tagged_captures [87,953 x 18]
##
##           tag fish_number species  cohort sample_number detection_date
##         (chr)       (chr)   (chr)   (chr)         (dbl)         (time)
## 1  00088cbed0           4     bkt 2012.00            73     2013-03-25
## 2  00088cbed2          20     bkt    2012            14     2012-09-07
```

```
## 3  00088cbed2          19     bkt      NA           15     2013-05-15
## 4  00088cbed3           2     bkt 2012.00           70     2012-06-07
## 5  00088cbed4           3     bkt      NA           73     2013-03-29
## 6  00088cbed4           7     bkt      NA           74     2013-06-25
## 7  00088cbed6           3     bnt      NA           70     2012-06-08
## 8  00088cbed6           2     bnt      NA           71     2012-09-27
## 9  00088cbed6           1     bnt      NA           73     2013-03-28
## 10 00088cbed7           9     bkt 2014.00           79     2014-09-17
## ..        ...         ...     ...     ...          ...            ...
## Variables not shown: season_number (dbl), drainage (chr), river (chr),
##   area (chr), section (chr), observed_length (dbl), observed_weight (dbl),
##   maturity (chr), sex (chr), survey (chr), sample_name (chr), comments
##   (chr)
```

Any functions that are placed after tbl and before collect are carried out in postgres, so it is wise to trim the data using these functions if the tables of interest are very large to minimize download times. A more thorough introduction to using this connection can be found in the dplyr database connection vignette.

'con' is an object that is used by the RPostgreSQL package. The dplyr connection is more user-friendly, but if details on how to use this connection are of interest, they can be found through the RPostgreSQL documentation.

Database structure

The database is relational, so not all of the information that you will need is going to be stored in a single table. For example the electrofishing data stored in the table data_tagged_captures only has data unique to a particular capture event reliably; the species, cohort, and other constant information associated with a tag number is stored in the table data_by_tag. Thus, a join of these two will be necessary to make the capture data useful.

Tables names that start with data_ are the ones that are meant for use. raw_ files are just imported with all columns as character and stored. tags_ files represent and intermediate stage at which columns are transformed to appropriate types but not yet filtered and organized for easy access.

Obtaining data using the built-in functions

There are a number of functions that grab, join, and manipulate data to try to reduce some of the data wrangling burden for common tasks. Postgres does not like capital letters in table or column names, so underscores are used on the database side; however, these functions convert those names to camelCase. Any column specification can be done using either under_scores or camelCase, which are converted with a helper function convertCase().

Creating a basic data.frame

The following code downloads data from the database and joins it to individual and sample specific tables to create a reasonable working data.frame. The defaults for functions in general tend to be designed to facilitate the creation of data for CJS models; for example, the default columns for createCoreData() do not include riverMeter, which would be absolutely necessary to make sense of stationaryAntenna data.

```
createCoreData(sampleType=c("electrofishing","stationaryAntenna"),
               columnsToAdd=c("observed_weight","riverMeter"),
               whichDrainage="west") %>% #downloads all data meeting the sample type and drainage speci
  addTagProperties() %>% #joins with data_by_tag
  filter(species=="bkt") %>% #filtering rows is just done manually
  addSampleProperties() #joins with data_seasonal_sampling
```

```
## Source: local data frame [820,817 x 15]
##
##           tag       detectionDate       river riverMeter sampleName
##         (chr)              (time)       (chr)      (dbl)      (chr)
## 1  00088cbed0 2013-03-25 00:00:00    wb obear         NA         84
## 2  00088cbed3 2012-06-07 00:00:00  west brook         NA         81
## 3  00088cbed4 2013-03-29 00:00:00  west brook         NA         84
## 4  00088cbed4 2013-06-25 00:00:00  west brook         NA         85
## 5  00088cbed7 2014-09-17 00:00:00    wb obear         NA         90
## 6  00088cbed7 2014-12-01 00:00:00    wb obear         NA         91
## 7  00088cbed9 2014-09-19 00:00:00  west brook         NA         90
## 8  00088cbed9 2014-10-17 20:22:33 wb mitchell    4797.25         NA
## 9  00088cbed9 2014-10-17 20:22:33 wb mitchell    4797.25         NA
## 10 00088cbed9 2014-10-24 15:34:31 wb mitchell    4797.25         NA
## ..        ...                 ...         ...        ...        ...
## Variables not shown: sampleNumber (dbl), section (chr), area (chr),
##   observedLength (dbl), observedWeight (dbl), species (chr), cohort (dbl),
##   year (dbl), season (dbl), proportionSampled (dbl)
```

Creating data for a CJS model in JAGS

The most complete data pipeline is the preparation of data to run CJS models in JAGS. The order here is important. createCmrData() filters to only include seasonal samples and then fills in times when a fish was not observed. It is therefore important to addSampleProperties() after this because there will be new rows added. The same goes for addEnvironmental(). createJagsData() turns the data.frame into a list of the data components that are likely to be used in a JAGS model and adds control structures (e.g., nEvalRows) designed for long format models (vectors rather than matrices of observations).

```
coreData<-createCoreData(sampleType="electrofishing") %>%
  addTagProperties() %>%
  dplyr::filter(species=="bkt") %>%
  createCmrData() %>%
  fillSizeLocation() %>% #assume fish grow linearly and stay put until they are found elsewhere
  addSampleProperties() %>%
  addEnvironmental(sampleFlow=T) %>%
  addKnownZ()
str(coreData)
```

```
## Classes 'tbl_df', 'tbl' and 'data.frame':    213010 obs. of  22 variables:
##  $ tag              : chr  "00088cbed0" "00088cbed0" "00088cbed0" "00088cbed0" ...
##  $ detectionDate    : POSIXct, format: "2013-03-25" "2013-06-06" ...
##  $ flowForP         : num  0.3201 0.1339 0.0408 0.0444 0.9933 ...
##  $ cohort           : num  2012 2012 2012 2012 2012 ...
##  $ sampleName       : chr  "84" "85" "86" "87" ...
##  $ sampleNumber     : num  73 74 75 76 77 78 79 80 81 82 ...
```

```
##  $ river            : chr  "wb obear" "wb obear" "wb obear" "wb obear" ...
##  $ section          : chr  "7" "7" "7" "7" ...
##  $ area             : chr  "trib" NA NA NA ...
##  $ observedLength   : num  62 96.3 106.8 111.3 113.6 ...
##  $ species          : chr  "bkt" "bkt" "bkt" "bkt" ...
##  $ enc              : num  1 0 0 0 0 0 0 0 0 0 ...
##  $ ageInSamples     : num  3 4 5 6 7 8 9 10 11 12 ...
##  $ sampleIndex      : num  53 54 55 56 57 58 59 60 61 62 ...
##  $ tagIndex         : num  1 1 1 1 1 1 1 1 1 1 ...
##  $ year             : num  2013 2013 2013 2013 2014 ...
##  $ season           : num  1 2 3 4 1 2 3 4 1 2 ...
##  $ proportionSampled: num  1 1 1 1 1 1 1 1 1 1 ...
##  $ lagDetectionDate : POSIXct, format: "2013-06-06" "2013-09-23" ...
##  $ meanTemperature  : num  8.671 15.569 8.932 0.903 9.288 ...
##  $ meanFlow         : num  0.38538 0.29172 0.00914 0.3132 0.58354 ...
##  $ knownZ           : num  1 NA NA NA NA NA NA NA NA NA ...
```

```
jagsData<-createJagsData(coreData)
str(jagsData)
```

```
## List of 35
##  $ proportionSampled: num [1:213010] 1 1 1 1 1 1 1 1 1 1 ...
##  $ intervalMeans    : num [1:4, 1:5] 0.000824 0.001172 0.000803 0.001393 0.000817 ...
##  $ dIntDaysMean     : num [1:16, 1:3] 1 1 1 1 2 2 2 2 3 3 ...
##   ..- attr(*, "dimnames")=List of 2
##   .. ..$ : NULL
##   .. ..$ : chr [1:3] "season" "river" "int"
##  $ nOut             : int 199030
##  $ nLastObsRows     : int 13980
##  $ lastObsRows      : int [1:13980] 11 25 36 41 42 56 61 74 79 84 ...
##  $ evalRows         : int [1:199030] 2 3 4 5 6 7 8 9 10 11 ...
##  $ nEvalRows        : int 199030
##  $ occ              : num [1:213010] 51 52 53 54 55 56 57 58 59 60 ...
##  $ nOcc             : int 63
##  $ firstObsRows     : int [1:13980] 1 12 26 37 42 43 57 62 75 80 ...
##  $ nFirstObsRows    : int 13980
##  $ nAllRows         : int 213010
##  $ season           : num [1:213010] 1 2 3 4 1 2 3 4 1 2 ...
##  $ nYears           : num 16
##  $ year             : num [1:213010] 14 14 14 14 15 15 15 15 16 16 ...
##  $ intervalDays     : num [1:213010] 0.000845 0.001262 0.00066 0.001574 0.000799 ...
##  $ tempSDDATA       : num [1:4, 1:5] 1.219 0.765 1.176 0.62 1.178 ...
##  $ tempMeanDATA     : num [1:4, 1:5] 10.2 16.21 8.44 1.55 9.65 ...
##  $ flowSDDATA       : num [1:4, 1:5] 0.165 0.136 0.239 0.123 0.165 ...
##  $ flowMeanDATA     : num [1:4, 1:5] 0.431 0.13 0.283 0.467 0.424 ...
##  $ flowDATA         : num [1:213010] 0.38538 0.29172 0.00914 0.3132 0.58354 ...
##  $ tempDATA         : num [1:213010] 8.671 15.569 8.932 0.903 9.288 ...
##  $ flowSd           : num [1:4, 1:5] 0.165 0.136 0.239 0.123 0.165 ...
##  $ flowMean         : num [1:4, 1:5] 0.431 0.13 0.283 0.467 0.424 ...
##  $ tempSd           : num [1:4, 1:5] 1.219 0.765 1.176 0.62 1.178 ...
##  $ tempMean         : num [1:4, 1:5] 10.2 16.21 8.44 1.55 9.65 ...
##  $ lengthSd         : num [1:4, 1:5] 19.3 18.4 18.5 17.1 16.5 ...
##  $ lengthMean       : num [1:4, 1:5] 143 150 145 143 140 ...
##  $ ind              : num [1:213010] 1 1 1 1 1 1 1 1 1 1 ...
```

```
##  $ lengthDATA       : num [1:213010] 62 96.3 106.8 111.3 113.6 ...
##  $ nRivers          : int 4
##  $ riverDATA        : num [1:213010] 4 4 4 4 4 4 4 4 4 4 ...
##  $ z                : num [1:213010] 1 NA NA NA NA NA NA NA NA NA ...
##  $ encDATA          : num [1:213010] 1 0 0 0 0 0 0 0 0 0 ...
```

Getting data for non-CJS applications

For the moment, the best way to get data for other applications is to just use the first few functions in the pipeline and trudge on manually from there.

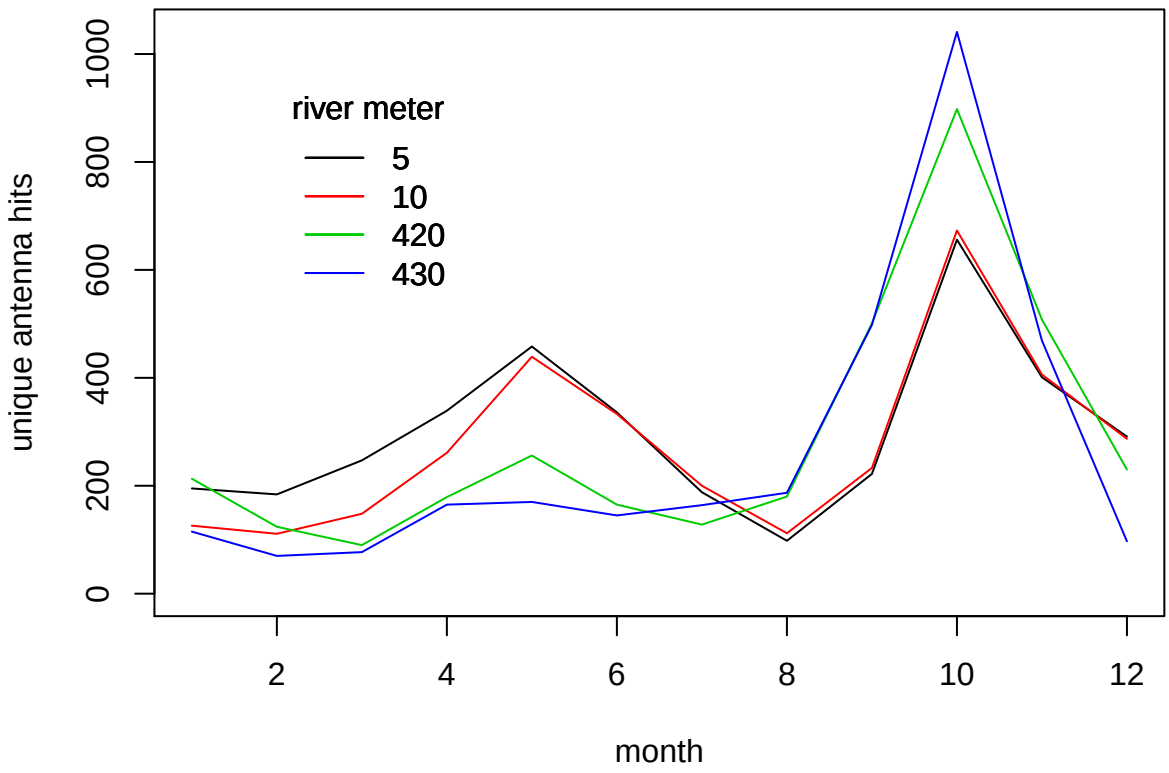
```
library(lubridate)
```

```
## Warning: package 'lubridate' was built under R version 3.2.3
```

```
antenna<-createCoreData(sampleType="stationaryAntenna",whichDrainage="stanley",
                        columnsToAdd="riverMeter") %>%
          addTagProperties() %>%
          filter(!is.na(riverMeter)) %>%
          mutate(month=month(detectionDate),
                 year=year(detectionDate)) %>%
          group_by(month,year,riverMeter) %>%
          summarize(nFish=length(unique(tag))) %>%
          ungroup() %>%
          group_by(month,riverMeter) %>%
          summarize(nFish=sum(nFish)) %>%
          ungroup() %>%
          arrange(riverMeter,month)
```

```
## Warning in createCoreData(sampleType = "stationaryAntenna", whichDrainage
## = "stanley", : column(s) sample_name, sample_number, section, area,
## observed_length do(es) not exist in any sampleType selected
```

```
plot(NA,xlim=c(1,12),ylim=c(0,max(antenna$nFish)),
     xlab="month",ylab="unique antenna hits")
for(s in unique(antenna$riverMeter)){
  points(nFish~month,data=antenna[antenna$riverMeter==s,],type='l',
       col=palette()[which(s==unique(antenna$riverMeter))])
  legend(2,950,unique(antenna$riverMeter),
         col=palette()[1:length(unique(antenna$riverMeter))],
         lty=1,bty='n',title="river meter")
}
```



New functions

There will be many applications, including most applications using antenna and acoustic data, where these functions will not suffice. Functional solutions to common problems would be useful additions to this package. We have tried to follow the basic form of having the first argument of each function be the output of the function before it in the pipeline. This facilitates the use of the %>% to pass data through.