modeid usage example

Duncan Procter

24th January 2018

Requirements

To use this example you will need quite a few packages installed. Going though it you can see where each is loaded. The modeid package is required throughout, and can be installed from github via a devtools function, install_github:

```r
install.packages("devtools","zoo","xgboost","moments","sp","rgdal","maptools","spatstat")
library(devtools)
install_github("dprocter/modeid",quiet=TRUE)
```

Data Processing

Let us assume:

1. You have raw ActiGraph accelerometer files, exported to .csv, with headers and without timestamps
2. You have corresponding GPS data

If either of these is untrue, but you'd like to make use of the algorithm, you probably need to contact the author, dprocter@gmail.com, and we will see what we can do

Processing the Accelerometer data

The first thing is to process the accelerometer data and summarise to epochs. This is quite a time consuming step, because a week of accelerometer data at 30Hz contains approx 19million rows of data, which we need to summarise. We call the GGIR package to calibrate the accelerometer data and assess non-wear time.

Here I have assigned accfile to the path to the accelerometer file on my computer, replace it is with the path to your accelerometer file

see ?process.acc for full details of the parameters. Briefly:

cutoff.method - the trimmming of accelerometer data, if you want it cut to only 7 days for example, here 1 means don't trim it, see ?process.acc for more options

epoch length - the length of epoch you want raw data summarised to, in seconds

acc.model - the accelerometer model, this package can also merge data from Actiheart data to GPS, but it will not be able to make model predictions to that data. If you have another brand of acceleormeter, contact the package author, we would be happy to edit the package to process your data.

raw - whether it is raw data, to predict the model to the data it needs to be raw

samples.per.second - the sampling rate, 30Hz is standard for Actigraph GT3x I think

nonwear - Whether you want non-wear time assessed. This will call the GGIR package, if you want to do this yourself using GGIR or some other means set nonwear=FALSE. This step is quite time consuming, but that is not surprising, it is a huge amount to calculate.

```r
library(modeid)

acc.data<-process.acc(accfile = accfile
                      ,participant.id = "id1"
                      ,cutoff.method = 1
                      ,epoch.length = 10
                      ,acc.model = "Actigraph"
                      ,raw=TRUE
                      ,samples.per.second = 30
                      ,nonwear=TRUE)
```

Merging accelerometer and GPS data

Next we need to merge the accelerometer data with gps data, to create a single data-set. To do this again,

British.time=TRUE means that the data is from the UK, so we need to convert UTC time from the GPS unit to British Summer time, during the appropriate months, so it matches the correct accelerometer data. If you have data from elsewhere, with more adjustments to include, email dprocter@gmail.com, I can take that into account and update the function

```r
merged.data<-gps.acc.merge(acc.data = acc.data
                          ,gpsfile = gpsfile
                          ,participant.id = "id1"
                          ,epoch.length = 10
                          ,british.time = TRUE
                          ,UTC.offset = 0)
```

Cleaning GPS data

There are several circumstances is which GPS data can be unreliable (usually caused by poor signal). Therefore we remove points we do not think we can trust.

We clean the data in 3 ways:

1. Using a speed cut-off, to remove implausibly high speed points
2. Using a Horisontal Dilution of Precision cut-off, to remove points where the satellites are aligned and so signal is poor
3. By removing points that are isolated, and thefore have no context

The following therefore marks all GPS data where speed is over 200kph, hdop is over 5, or there are less than 3 points within 5 minutes (2.5 minutes before and after the points, inluding the point itself, therefore 2 neighbours) as NA.

The *data.loss.gps* function tell you how many points are removed at each level of processing

```r
data.loss.gps(speed.cutoff = 200
              ,hdop.cutoff = 5
              ,neighbour.number = 3
              ,neighbour.window = 300
              ,epoch.length = 10
              ,dataset = merged.data)
```

```
##               labels data.amounts data.removed
## 1 total.dataset.size        69120            0
```

```
## 2   invalid.gps.data        27913        41207
## 3       no.neigbours        27900           13
## 4       excess.speed        27900            0
## 5        poor.signal        27611          289
```

```
merged.data<-gps.cleaner(speed.cutoff = 200
                         ,hdop.cutoff = 5
                         ,neighbour.number = 3
                         ,neighbour.window = 300
                         ,epoch.length = 10
                         ,dataset = merged.data)
```

Calculating distance to train lines

This doesn't cover getting the neccessary data on train lines. As a start point, if you're in a UK institution there is lots of data freely available on Digimap. Another option is to use OpenStreetMap data, which you can access directly from R using the *osmdata* package, which has a good vignette here: https://cran.r-project.org/web/packages/osmdata/vignettes/osmdata.html

If you do not care about train travel, you can just create an empty variable called *near.train* which is entirely NA. Xgboost can predict ot data with NAs, and the train travel will most-likely come out as vehicle travel because of the similarity in speed.

Here is an example of how you can extract train data for London using osmdata. We extract three separate slices of train data, rail lines, light rail lines and subway/underground lines, then combine them. If it is quite a large area that you need to extract (e.g. the data for our study covers most of England and Wales), this will take a long time.

```
install.packages("osmdata", "magrittr", "raster")
```

```
library(osmdata)
```

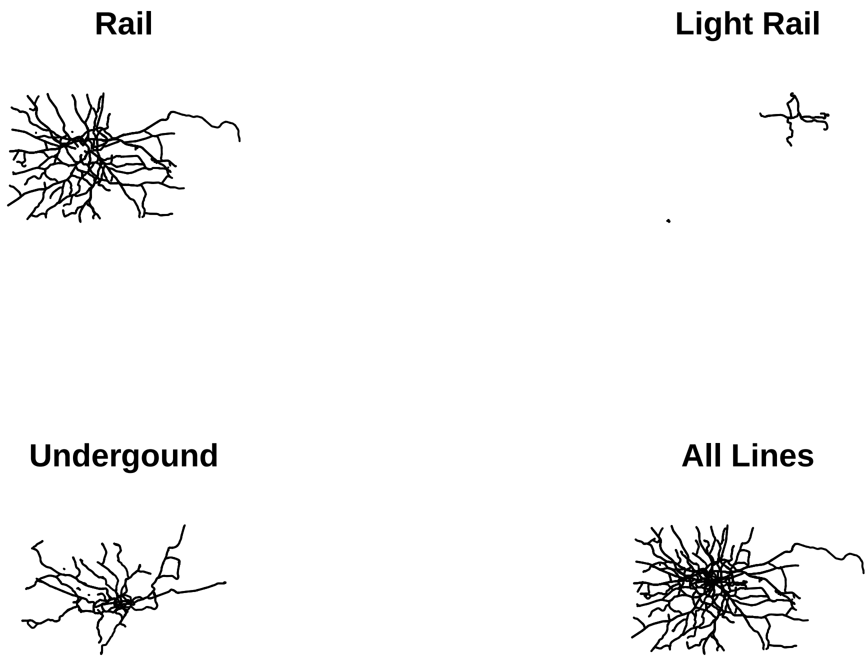
```
## Data (c) OpenStreetMap contributors, ODbL 1.0. http://www.openstreetmap.org/copyright
```

```
library(magrittr)
library(sp)
library(raster)
```

```
##
## Attaching package: 'raster'

## The following object is masked from 'package:magrittr':
##
##     extract
```

```
p<- opq(bbox="London, UK") %>%
  add_osm_feature(key = 'railway',value="rail")
q<- opq(bbox="London, UK") %>%
  add_osm_feature(key = 'railway',value="light_rail")
r<- opq(bbox="London, UK") %>%
    add_osm_feature(key = 'railway',value="subway")

par(mfrow=c(2,2))
p.train<-osmdata_sp(p)
p.train<-p.train$osm_lines
plot(p.train, main="Rail")
q.train<-osmdata_sp(q)
```

```
q.train<-q.train$osm_lines
plot(q.train, main="Light Rail")
r.train<-osmdata_sp(r)
r.train<-r.train$osm_lines
plot(r.train, main="Undergound")

london.train.data <- do.call(bind, list(q.train,p.train,r.train))
plot(london.train.data, main="All Lines")
```



For our project, we used a combination of OS Opendata and meridian 2 rail networks, which was combined in ArcGIS. Here I import that data into R using the the *sp* and *rgdal* packages, then convert the *SpatialLinesDataFrame* into a *psp* (a line segment pattern) so we can use the *spatstat* function *nncross* to measure distance from each point to the nearest train line. The near.train function takes a merged dataset and train line data and calls the *nncross* functions to measure distance from each point to the nearest trainline.

```
library(sp, quietly = TRUE)
library(maptools, quietly = TRUE)
```

```
## Checking rgeos availability: TRUE
```

```
library(rgdal, quietly = TRUE)
```

```
## rgdal: version: 1.2-16, (SVN revision 701)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 2.2.0, released 2017/04/28
##  Path to GDAL shared files: C:/Duncan/R libraries/rgdal/gdal
##  GDAL binary built with GEOS: TRUE
```

```
##  Loaded PROJ.4 runtime: Rel. 4.9.3, 15 August 2016, [PJ_VERSION: 493]
##  Path to PROJ.4 shared files: C:/Duncan/R libraries/rgdal/proj
##  Linking to sp version: 1.2-5
```

```
library(spatstat, quietly = TRUE)
```

```
##
## Attaching package: 'nlme'

## The following object is masked from 'package:raster':
##
##     getData

##
## spatstat 1.54-0       (nickname: 'Vacuous Mission Statement')
## For an introduction to spatstat, type 'beginner'

##
## Note: R version 3.3.3 (2017-03-06) is more than 9 months old; we strongly recommend upgrading to the

##
## Attaching package: 'spatstat'

## The following objects are masked from 'package:raster':
##
##     area, rotate, shift
```

```
train.lines<-readOGR("C:/Duncan/Train lines","all_train_lines")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "C:/Duncan/Train lines", layer: "all_train_lines"
## with 38316 features
## It has 8 fields
## Integer64 fields read as strings:  OBJECTID CODE
```

```
train.coords<-coordinates(train.lines)
max.x<-max(unlist(lapply(train.coords,FUN=function(x){x[[1]][,1]})))
max.y<-max(unlist(lapply(train.coords,FUN=function(x){x[[1]][,2]})))
min.x<-min(unlist(lapply(train.coords,FUN=function(x){x[[1]][,1]})))
min.y<-min(unlist(lapply(train.coords,FUN=function(x){x[[1]][,2]})))

train.win<-owin(xrange=c(min.x,max.x),yrange=c(min.y,max.y))

train.psp<-as.psp(train.lines,W=train.win)
```

```
## Warning in as.psp.SpatialLinesDataFrame(train.lines, W = train.win): 7
## columns of data frame discarded
```

```
merged.data<-near.train(dataset = merged.data
                        , trainline.psp = train.psp
                        , trainline.p4s = proj4string(train.lines))
```

```
## Warning: data contain duplicated points
```

Distance 1 minute away

When not travelling, people stay in one spot. To take this into consideration we include distance moved in the next minute and distance moved in the last minute. Both are included so that we can detect no movement both just as you stopped and just before you start moving too.

```r
merged.data$dist.next.min<-distance.moved(dataset = merged.data,
                                          last=FALSE,
                                          time.window = 60,
                                          epoch.length = 10)

merged.data$dist.last.min<-distance.moved(dataset = merged.data,
                                          last=TRUE,
                                          time.window = 60,
                                          epoch.length = 10)
```

Calculating moving windows

None of the previous functions remove invalid data from the dataset, they only set the relevant accelerometer or GPS variables as NA when we have reason to think they are untrustworthy. As a result the dataset is a continuous set of epochs from the start to end of the data. We can therefore treat a moving window across the cleaned merged dataset as a time window, as long as we allow for how long each epoch represents.

To calculate moving windows we use the *zoo* package, and particularly the *rollapply* function. Rollapply allows us to specify a function and then apply it across a window. We use a width of four minutes, centered on the point of interest. This is contained within the function *rollav.calc*, which takes a processed dataset, then calculates the necessary moving windows to fit a model.

```r
rollavs<-rollav.calc(dataset=merged.data)
```

Predicting to the example data

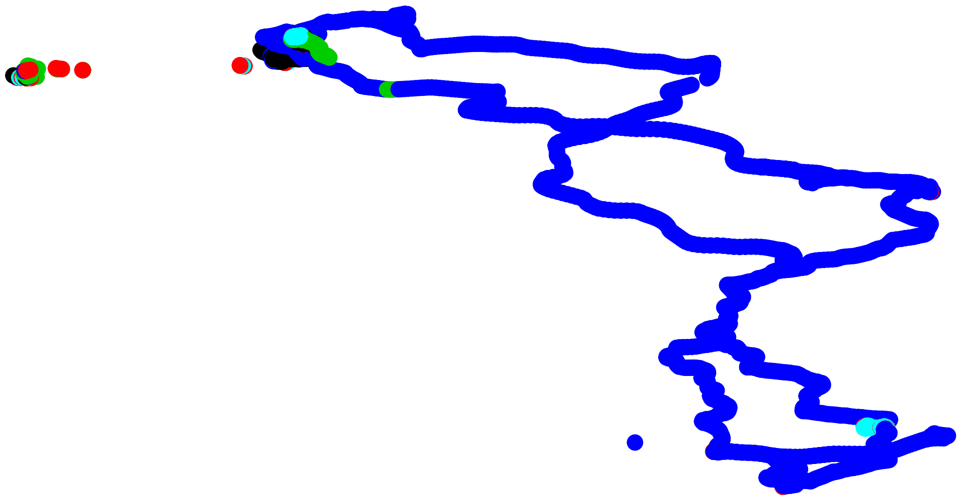
Prediction to participants is simple once you have calculated the rolling means, here we predict to this example file and then plot the points on a map coloured by travel mode. I have given minimal detail on where this is or a full legend, because this data is from a participant, and so I cannot make the coordinates freely available.

fitted.fullmod is the fitted model from our paper, which is cinluded within this package

```r
rollavs$pred.mode<-predict(fitted.fullmod,newdata = as.matrix(pred.data(rollavs)),type="response")
rollavs$pred.mode<-factor(rollavs$pred.mode, labels=c("cycle","stat","train","vehicle","walk"))
summary(rollavs$pred.mode)
```

```
##   cycle    stat   train vehicle    walk
##    1027   22308   40137    2855    2793
```

```r
plot(rollavs$easting,rollavs$northing,axes=FALSE,xlab="",ylab="",col=rollavs$pred.mode,pch=19)
```



Training data

We have privided an anonymised version of our training dataset, so you can see how we use it, and if you would like, fit different models to it:

```
training.data<-train.data
names(training.data)
```

```
##  [1] "id"               "day"              "date.time"
##  [4] "ax1.mad.4min"     "ax1.c90.4min"     "ax1.c10.4min"
##  [7] "ax1.skew.4min"    "ax1.kurt.4min"    "ax2.mad.4min"
## [10] "ax2.c90.4min"     "ax2.c10.4min"     "ax2.skew.4min"
## [13] "ax2.kurt.4min"    "ax3.mad.4min"     "ax3.c90.4min"
## [16] "ax3.c10.4min"     "ax3.skew.4min"    "ax3.kurt.4min"
## [19] "ax1.fft.4min"     "ax2.fft.4min"     "ax3.fft.4min"
## [22] "spd.mean.4min"    "spd.sd.4min"      "spd.c10.4min"
## [25] "spd.c90.4min"     "sumsnr.4min"      "near.train.4min"
## [28] "dist.next.4min"   "dist.last.4min"   "abs.acc.mean.4min"
## [31] "acc.sd.4min"      "lowsp.prop.4min"  "true.mode"
## [34] "cv.marker"        "true.mode.bus"
```

Replicating our cross.validation

The function *cross.validator* fits a number of xgboost models to the subsets of the data you specify.

As you can see below you must first provide a dataset free from NAs, so we na.omit to remove NAs

cross.validator then requires the NA free dataset, the label variable (in our case the true mode), and a marker to denote the cross-validation subsets. We simply randomly assigned each participant to one of 5 subsets, see ?sample for random number selection

cross.validator currently has horrible looking output, which consists of 3 columns of lists (the fitted models, the confusion matrices and model accuracy scores). The rows correspond to the cross-validation subsets, so in our example there will be 5. This will be tidied in future versions.

We use set.seed to give the randomness a start point and ensure output is replicable and set verbose=0 so that you don't get all of the xgboost output in this document, set verbose=1 if you want the training error per iteraction output.

eta sets how conservative the algorithm should be. Here we set it lower than usual, at 0.1 (default 0.3), to avoid over-fitting to the training data.

subsample = 0.2 means that we feed xgboost 20% of each cross-validation subset for each iteration, selected at random. This is essentially a second level of cross-validation, again, avoiding overfitting to the trianing data

Threads = 8 allows xgboost to use parralel computation, in the PC I ran this on I can run 8 threads at once so threads=8. If you have no idea what this means set threads as the number of cores your computer has, or 1 to not use parralel computation.

nrounds=200 means run 200 iterations per cross-validation subset. This is quite a high number, because we have set quite conservative parameters to stop over-fitting, so it needs time to optimally fit the model

gamma=10 is another parameter to prevent over-fitting and make the algorithm conservative. default=1, we set it higher to make it more conservative. See ?xgboost for details.

```r
train.for.cv<-training.data
train.for.cv<-na.omit(train.for.cv)

cv.model<-cross.validator(training.data = pred.data(train.for.cv)
                          , label = train.for.cv$true.mode
                          , cv.marker = train.for.cv$cv.marker
                          , seed=315
                          , method = "xgboost"
                          , eta = 0.1
                          , subsample = 0.2
                          , threads=8
                          , nrounds=200
                          , gamma=10
                          , verbose=0)

overall.acc(cvd.model = cv.model
        , full.dataset = training.data)
```

```
## [[1]]
##         cycle  stat train vehicle walk
## cycle    8078    18     4     311   23
## stat       49 45362    97     185  740
## train       3    10 12390      84   19
## vehicle   244   160   101   20964   17
## walk        8   321    24      36 9139
## error       0     0     0       0    0
##
## [[2]]
```

```
##     modes      ppv sensitivity      npv specificity accuracy f1.score
## 1   cycle 95.77899    96.37318 99.66205    99.60447 99.32918 96.07517
## 2    stat 97.69345    98.89037 99.02029    97.96062 98.39410 98.28826
## 3   train 99.07245    98.20862 99.73685    99.86476 99.65239 98.63864
## 4 vehicle 97.57051    97.14551 99.19897    99.32037 98.84334 97.35754
## 5    walk 95.91730    91.96015 99.10082    99.56020 98.79252 93.89705
```