gibbonR: An R package for classification, detection and visualization of acoustic signals using machine learning

Dena J. Clink & Holger Klinck

2019-05-20

PACKAGE DESCRIPTION

The field of bioacoustics is inherently multidisciplinary and relies on computer scientists, engineers, and ecologists. This package is directed towards ecologists who are interested in incorporating bioacoustics into their research, but may not have the skills or training. The goal for the creation of this package was to make commonly used signal processing techniques and various machine learning algorithms readily available for anyone interested in using bioacoustics in their research.

GETTING STARTED

First you need to install the package using the following lines of code:

```
install.packages(c("coda","mvtnorm","devtools","loo"))
library(devtools)
devtools::install_github("DenaJGibbon/gibbonR")
```

Then load the library

```
library(gibbonR)
```

This vignette delibrately includes some errors that will stop the functions from running to show common errors.The option below removes the "Show Traceback" option in RStudio.

```
options(error = NULL)
```

Set the directory where .wav files of detected sound events will be written

```
output.dir <- "/Users/denasmacbook/Desktop/output.test"
```

DATA PREPARATION

The sound files that we need to work through these examples are stored on Github. The code below will walk you through the steps of downloading the files.

```
# You need to tell R where to store the zip files on your computer.
destination.file.path.zip <-
  "/Users/denasmacbook/Desktop/BorneoExampleData.zip"

# You also need to tell R where to save the unzipped files
destination.file.path <- "/Users/denasmacbook/Desktop/BorneoExampleData"

# This function will download the data from github
utils::download.file("https://github.com/DenaJGibbon/BorneoExampleData/archive/master.zip",
                     destfile = destination.file.path.zip)
```

```
# This function will unzip the file
utils::unzip(zipfile = destination.file.path.zip,
             exdir = destination.file.path)

# Examine the contents
list.of.sound.files <- list.files(paste(destination.file.path,
                                        "BorneoExampleData-master", "data", sep =
                                          "/"),
                                  full.names = T)
list.of.sound.files
```

Use this function to read in the .RDA file and save it as an R object from https://stackoverflow.com/questions/5577221/how-can-i-load-an-object-into-a-variable-name-that-i-specify-from-an-r-data-file

```
loadRData <- function(fileName) {
  #loads an RData file, and returns it
  load(fileName)
  get(ls()[ls() != "fileName"])
}
```

This function will load the entire list of r data files

```
list.rda.files <- list()
for(x in 1:length(list.of.sound.files)){
  list.rda.files[[x]] <-  loadRData(list.of.sound.files[[x]])
}
```

Assign each rda an informative name

```
gibbon.females <- list.rda.files[[1]]
multi.class.list <- list.rda.files[[2]]
listBorneoSounds <- list()
  for (x in 1:5){
  listBorneoSounds[x] <- list.rda.files[[x+2]]
  }
training.MFCC.long <- list.rda.files[[8]]
```

Lists of sound files for training

Load .wav file data which is structured as a list with the name of the sound file as the first element and the .wav file as the second element. Classes are identified in the file names to the left of the "_"

```
# We will just check the first element for each list
gibbon.females[[1]]
multi.class.list[[1]]
```

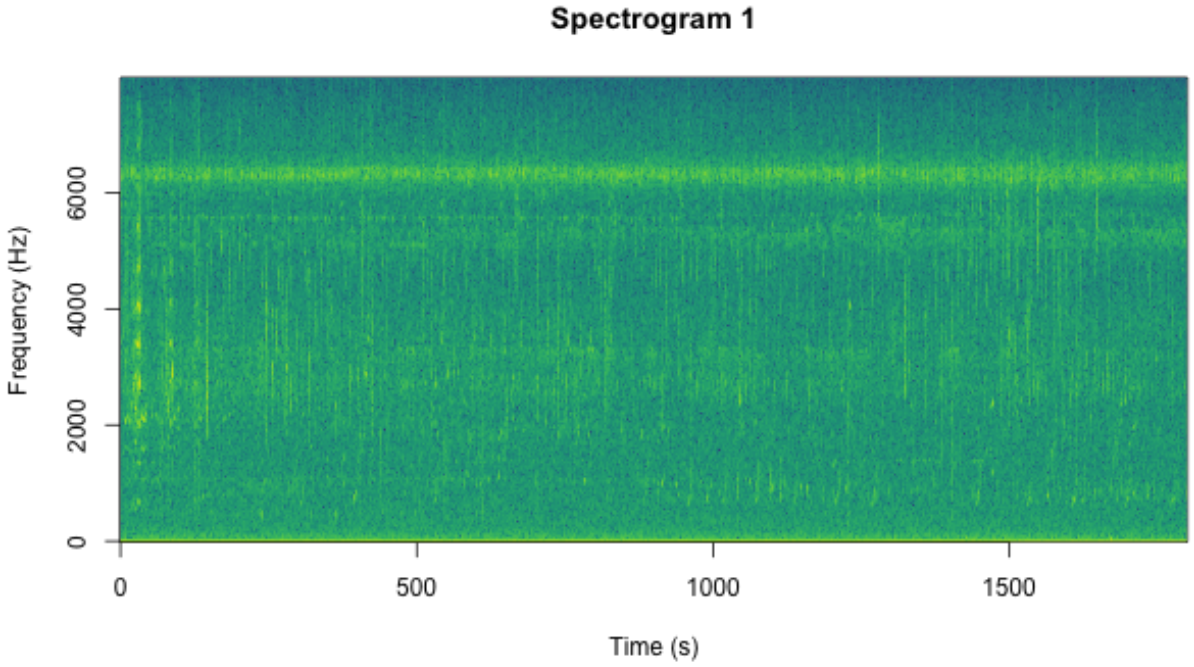
Alternatively, if you want to use your own training data you can load your own data into the correct list format using the following code.

```
# First, set the working directory where your .wav files are stored
setwd("file_path_to_wav_files")

# Create empty list for the .wav list
list.of.sound.files <- list()

# Create a list of .wav files in the above specified directory
```

```
list.wav.file.names <- list.files()

# Loop to read in all .wav files in directory and add the file names and .wav files to a list
for (x in 1:length(list.wav.file.names)) {
  tmp.wav <- tuneR::readWave(list.wav.file.names[x])
  list.of.sound.files[[x]] <- list(list.wav.file.names[x], tmp.wav)
}
```

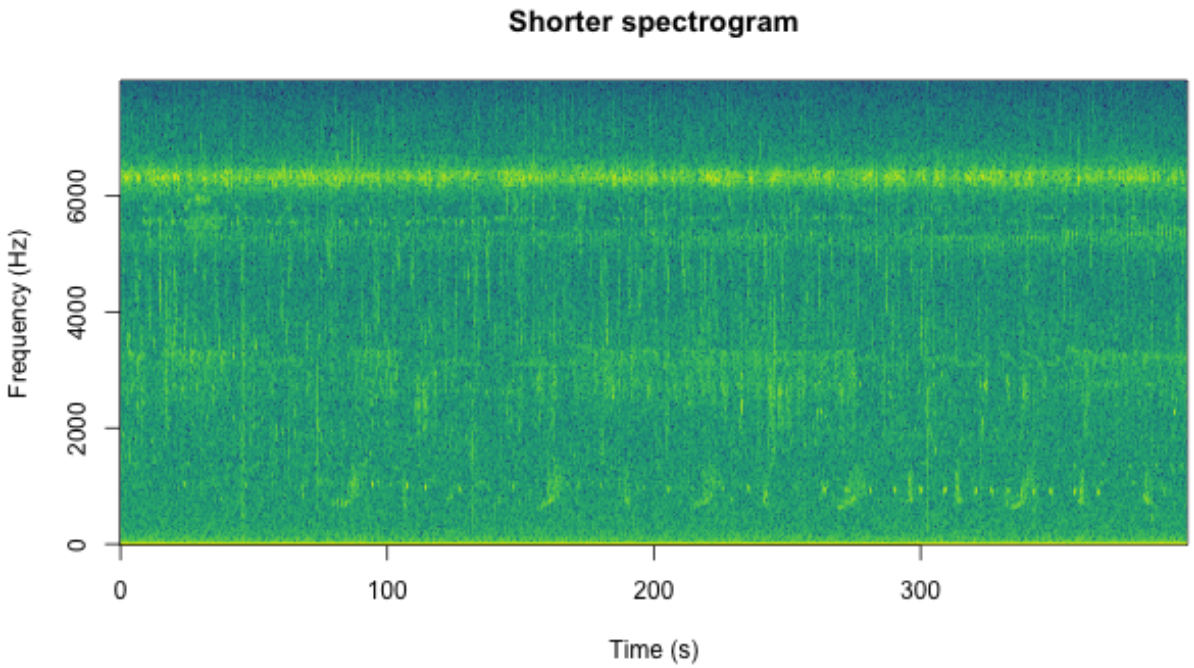
If you want to produce a spectrogram for each of the longer Borneo .wav files you can do that with the following code, just change "n.spectrograms" to the desired number of spectrograms

```
n.spectrograms <- 1
for (x in 1:n.spectrograms) {
  wav.1 <- listBorneoSounds[[x]]
  temp.spec <-
    signal::specgram(wav.1@left,
                     Fs = wav.1@samp.rate,
                     n = 512,
                     overlap = 0)
  temp.plot <-
    plot(
      temp.spec,
      xlab = "Time (s)",
      ylab = "Frequency (Hz)",
      col = viridis::viridis(512),
      useRaster = TRUE,
      main = paste("Spectrogram", x)
    )

}
```



We will focus on one Borneo .wav file to start, for simplicity and to reduce computing time– the chosen file has gibbon calls

```
wav.1 <- listBorneoSounds[[1]]
wav.1 <- seewave::cutw(wav.1, from = 800, to=1200, output = "Wave")

temp.spec <-
    signal::specgram(wav.1@left,
                     Fs = wav.1@samp.rate,
                     n = 512,
                     overlap = 0)
  temp.plot <-
    plot(
      temp.spec,
      xlab = "Time (s)",
      ylab = "Frequency (Hz)",
      col = viridis::viridis(512),
      useRaster = TRUE,
      main = paste("Shorter spectrogram")
    )
```



TRAINING AND TESTING MACHINE LEARNING ALGORITHMS

Here, our goal is to see how well classes of acoustic signals can be distinguished using different machine learning learning algorithms.We start by extracting MFCCs for each of the sound events. We will extract Mel-frequency cepstral coeffcients (MFCC) averaged over time windows.This is because the duration of the sound events is variable, and we require a feature vector of equal length for each call.

```
training.MFCC <- calcMFCC(
  list.wav.files = multi.class.list,
  n.window = 9,
  n.cep = 12,
  min.freq = 0,
  max.freq = 2000,
  feature.red = FALSE
)

# Check structure of dataframe
str(training.MFCC)
```

In some cases we may want to reduce the number of MFCCs, and one way to do this is via recursive feature selection which is an iterative process wherein predictors are ranked, and the less important features are subsequently eliminated, until a subset of predictors is identified that can produce an accurate model

```
# Recursive feature selection
#If feature.red = TRUE this does recursive feature selection

training.MFCC.rfe <- calcMFCC(
  list.wav.files = multi.class.list,
  n.window = 10,
  n.cep = 12,
  min.freq = 0,
  max.freq = 2000,
  feature.red = TRUE
)

# If feature.red = TRUE we get a smaller set of informative features
str(training.MFCC.rfe)
```

Visualization of call classes. First, create a biplot based on linear discriminant function analysis of MFCC features for full feature set NOTE: PCA can not be calculated as number of features exceeds number of observations in this example.

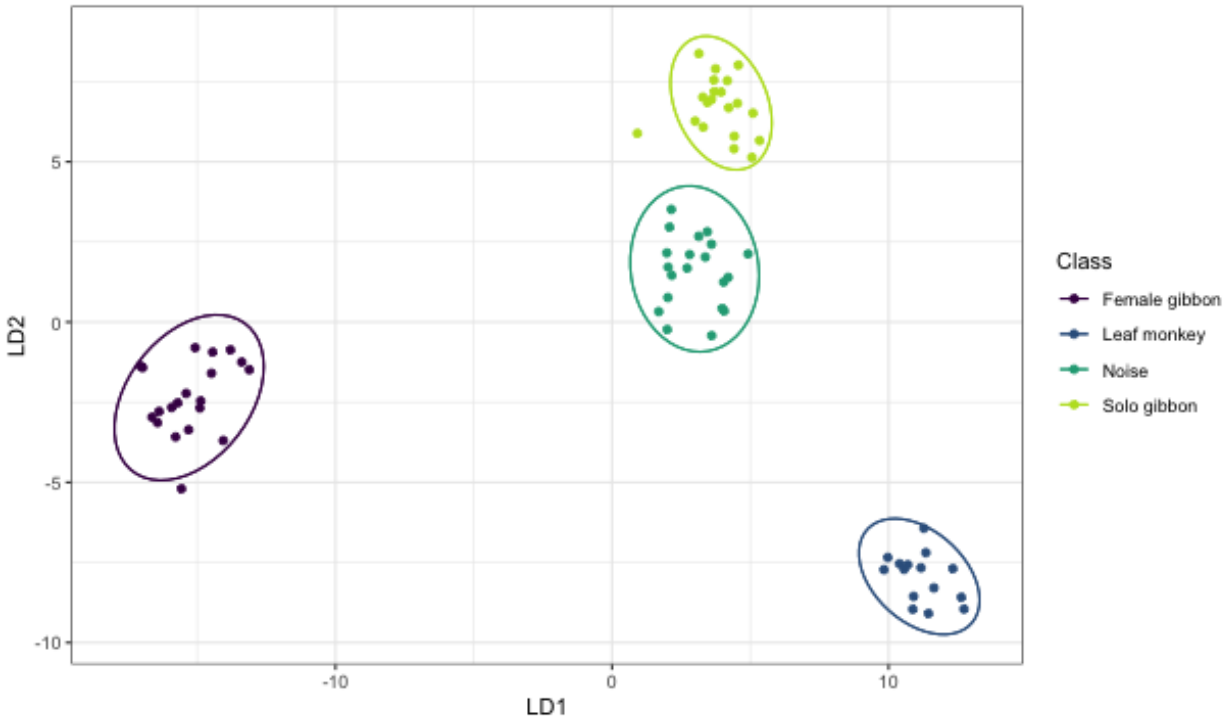
```
biplotGibbonR(training.MFCC,
              classification.type = "LDA",
              class.labs = F)
```

The default output does not have nice class names, so we can change them using the following code

```
training.MFCC$class <- plyr::revalue(training.MFCC$class, c("female.gibbon"="Female gibbon", "solo.gibb
                                "noise"="Noise"))
```

Then we create the biplot

```
# Create biplot based on LDA of MFCC features for full feature set
biplotGibbonR(training.MFCC,
              classification.type = "LDA",
              class.labs = F)
```



We can also create a biplot using principal component analysis

```
# Create biplot based on either PCA or LDA of MFCC features for reduced feature set
biplotGibbonR(training.MFCC.rfe,
              classification.type = "PCA",
              class.labs = F)
```

LINEAR DISCRIMINANT FUNCTION ANALYSIS

Check how well classes can be distinguished using leave-one-out cross-validation

```
# Check how well classes can be distinguished using leave-one-out cross-validation
output.lda <- trainLDA(feature.df = training.MFCC.rfe, CV = TRUE)
output.lda$correct.prob
```

Check how well classes can be distinguished using separate training and test datasets with full features

```
output.lda.full <-
  trainLDA(
    feature.df = training.MFCC,
    train.n = 0.8,
    test.n = 0.2,
    CV = FALSE
  )
##> Warning in lda.default(x, grouping, ...): variables are collinear
output.lda.full$correct.prob
```

This will output a warning that variables are collinear, which means that certain features are correlated. This will not negatively impact your classification ability, but shows that more features is not always neccesarily better.

Check how well classes can be distinguished using separate training and test datasets with reduced features

```
output.lda.red <-
  trainLDA(
    feature.df = training.MFCC.rfe,
    train.n = 0.8,
    test.n = 0.2,
    CV = FALSE
  )
output.lda.red$correct.prob
```

SUPPORT VECTOR MACHINES

Check how well classes can be distinguished using separate training and test datasets with full features

```
svm.output.full <-
  trainSVM(
    feature.df = training.MFCC,
    tune = "TRUE",
    train.n = 0.8,
    test.n = 0.2,
    cross = 5
  )
svm.output.full$correct.prob
```

Check how well classes can be distinguished using separate training and test datasets with reduced features

```
svm.output.red <-
  trainSVM(
    feature.df = training.MFCC.rfe,
    tune = "TRUE",
    train.n = 0.8,
    test.n = 0.2,
    cross=5
  )
svm.output.red$correct.prob
```

GAUSSIAN MIXTURE MODELS

Check how well classes can be distinguished using separate training and test datasets with full features

```
gmm.output.full <-
  trainGMM(feature.df = training.MFCC,
             train.n = 0.8,
             test.n = 0.2)
gmm.output.full$correct.prob

# NOTE: This will return an error as the model will fail to converge, which is related to the high numb
```

Check how well classes can be distinguished using separate training and test datasets with reduced features

```
gmm.output.red <-
  trainGMM(feature.df = training.MFCC.rfe,
             train.n = 0.8,
             test.n = 0.2)
gmm.output.red$correct.prob
```

NEURAL NETWORKS

Check how well classes can be distinguished using separate training and test datasets with full features

```
nn.output.full <-
  trainNN(feature.df = training.MFCC,
            train.n = 0.8,
            test.n = 0.2)
nn.output.full$correct.prob
```

Check how well classes can be distinguished using separate training and test datasets with reduced features

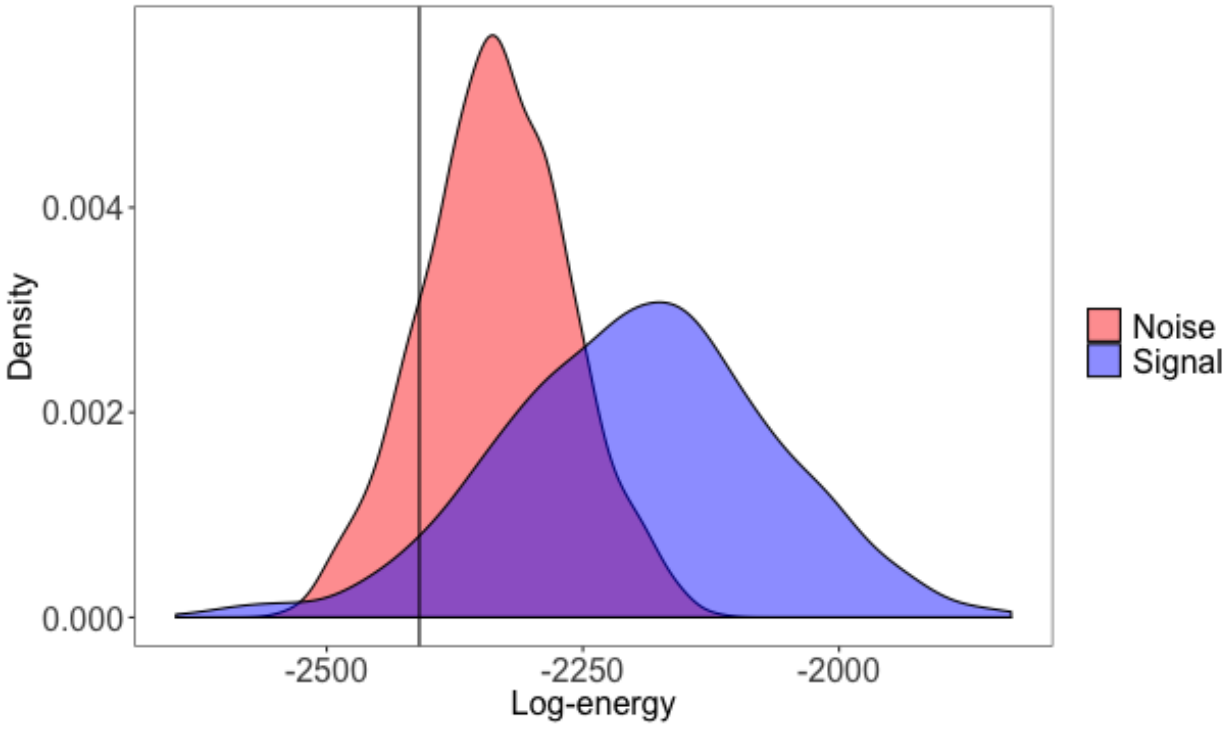
```
nn.output <-
  trainNN(feature.df = training.MFCC.rfe,
            train.n = 0.8,
            test.n = 0.2)
nn.output$correct.prob
```

AUDIO SEGMENTATION

AUDIO SEGMENTATION WITH GAUSSIAN MIXTURE MODELS

We will use two different approaches to identify potential sound events from long-term recordings. The first does not require the user to input any training data, and uses a bi-Gaussian mixture model to create two distributions based on the energy of a spectrogram of the recording. One distribution is the "noise" and the other is "signal". The user can define the cutoff threshold for "noise", with the most commonly used threshold being the intersection point between the two distributions. When density "plot=TRUE" this will return a plot of the noise and signal distributions, along with the location of the cutoff threshold.

```
# Audio Segementation using Gaussian mixture models
gmm_audio_output <-
  audioSegmentGMM(
    wav.file = wav.1,
    which.quant = "low.quant",
    output.type = "wav",
    min.signal.dur=1,
    low.quant.val = 0.15,
    n.window = 3,
    density.plot = TRUE,
    output.dir = output.dir
  )
```

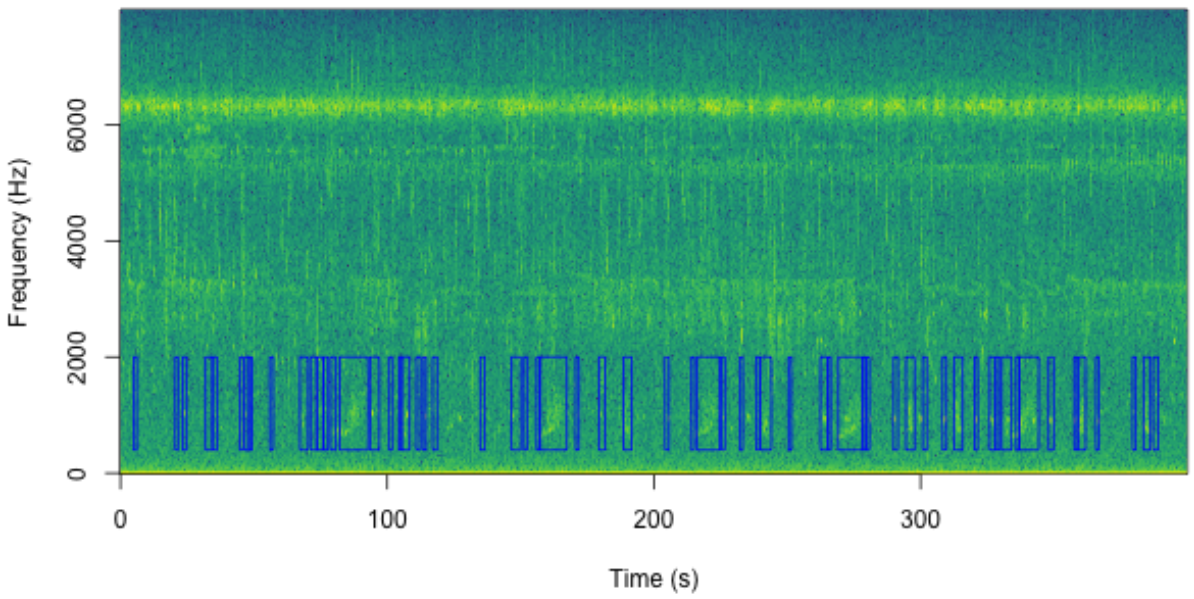



```
# The function provides an event table with the start and end time of each potential sound event
gmm_audio_output$sound.event.df

# We assign the dataframe to a temporary R object
temp.sound.event.df <- gmm_audio_output$sound.event.df
```

We can then create a spectrogram and outline the sound events on the spectrogram

```
# First calculate the spectrogram matrix
temp.spec <-
  signal::specgram(wav.1@left,
                   Fs = wav.1@samp.rate,
                   n = 512,
                   overlap = 0)

# Then plot the spectrogram
plot(
  temp.spec,
  xlab = "Time (s)",
  ylab = "Frequency (Hz)",
  col = viridis::viridis(512),
  useRaster = TRUE
)

# And add boxes for the identified sound events
for (x in 1:nrow(temp.sound.event.df)) {
  rect(temp.sound.event.df[x, 2],
       400,
       temp.sound.event.df[x, 3],
       2000,
```

```
        border = "blue")
}
```



AUDIO SEGMENTATION WITH SUPPORT VECTOR MACHINES

This audio segmentation technique requires the user to input labeled training data. For this example, we will calculate Mel-frequency cepstral coefficients for each of the .wav files in the training dataset. Each .wav file is of different duration, so there will be a variable number of time windows for the MFCC calculations. For this method, we will assign each time window a class value by specifying win.avg = "FALSE", and we will not average over time windows.

```
trainingdata <- calcMFCC(
  list.wav.files = multi.class.list,
  win.avg = "FALSE",
  win.hop.time = 0.25,
  n.window = 9,
  n.cep = 12,
  min.freq = 400,
  max.freq = 2000
)
```

Check structure of the training data; for each of the specified MFCCs (in this case n.cep=12) there will be a single value. Each column represents the MFCCs calculated for a particular time window. The default for the MFCC calculation sets overlap to zero, making time calculations relatively straight forward.

```
str(trainingdata)
```

We will then convert specific class names to more general names as we are simply looking for any gibbon call.

```
trainingdata$class <-
  plyr::revalue(trainingdata$class,
                c("female.gibbon" = "gibbon", "solo.gibbon" = "gibbon"))
```

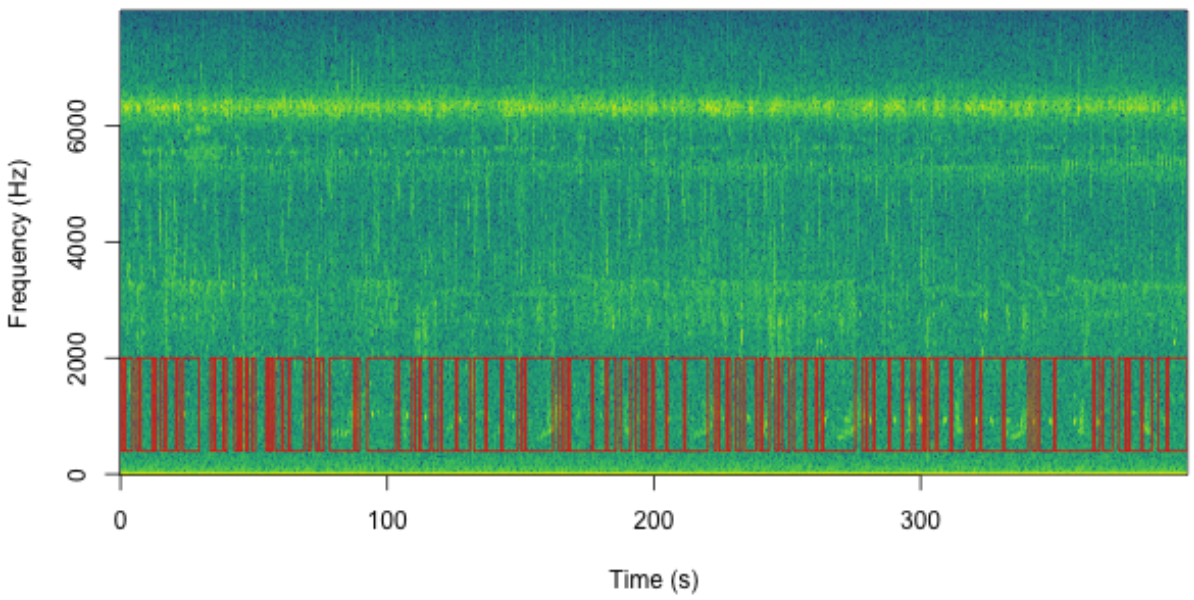
We will then use the audioSegmentSVM function to do audio segmentation on the sound file using support-vector machine

```
# NOTE: On large .wav files this will take a substantial amount of time to run.
svm_audio_output <- audioSegmentSVM(
  wav.file = wav.1,
  target.signal = "gibbon",
  min.signal.dur = 1,
  min.freq = 400,
  max.freq = 2000,
  prob.signal = 0.65,
  trainingdata = trainingdata,
  output.dir = output.dir,
  writetodir = FALSE
)

# This function returns three objects, the accuracy of the trained SVM, a detection dataframe by time w
svm_audio_output$svm.accuracy
svm_audio_output$detection.df
svm_audio_output$sound.event.df
```

We can then visualize our results.

```
# We assign the sound event dataframe to a temporary R object
temp.sound.event.df <- svm_audio_output$sound.event.df

# We can then create a spectrogram and outline the sound events on the spectrogram
# First calculate the spectrogram matrix
temp.spec <-
  signal::specgram(wav.1@left,
                   Fs = wav.1@samp.rate,
                   n = 512,
                   overlap = 0)

# Then plot the spectrogram
plot(
  temp.spec,
  xlab = "Time (s)",
  ylab = "Frequency (Hz)",
  col = viridis::viridis(512),
  useRaster = TRUE
)

# And add boxes for the identified sound events
for (x in 1:nrow(temp.sound.event.df)) {
  rect(temp.sound.event.df[x, 2],
       400,
       temp.sound.event.df[x, 3],
       2000,
       border = "red")
```

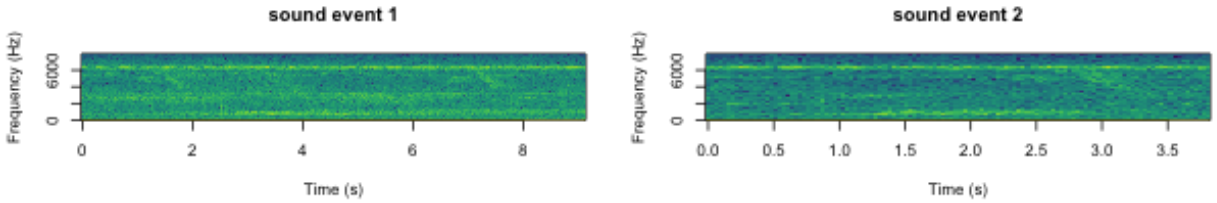
```
}
```



VISUALIZATION OF SOUND EVENTS

This function will plot multiple spectrograms on a single plot to facilitate inspection of sound events.

```
# NOTE: .wav files of long duration will process very slowly or not at all-- this function is recommend

sound.event.table <-
  plotSoundevents(input.dir = output.dir, n.soundevents =2,
                  nrow = 3,
                  ncol = 2)
```



CLASSIFICATION OF SOUND EVENTS

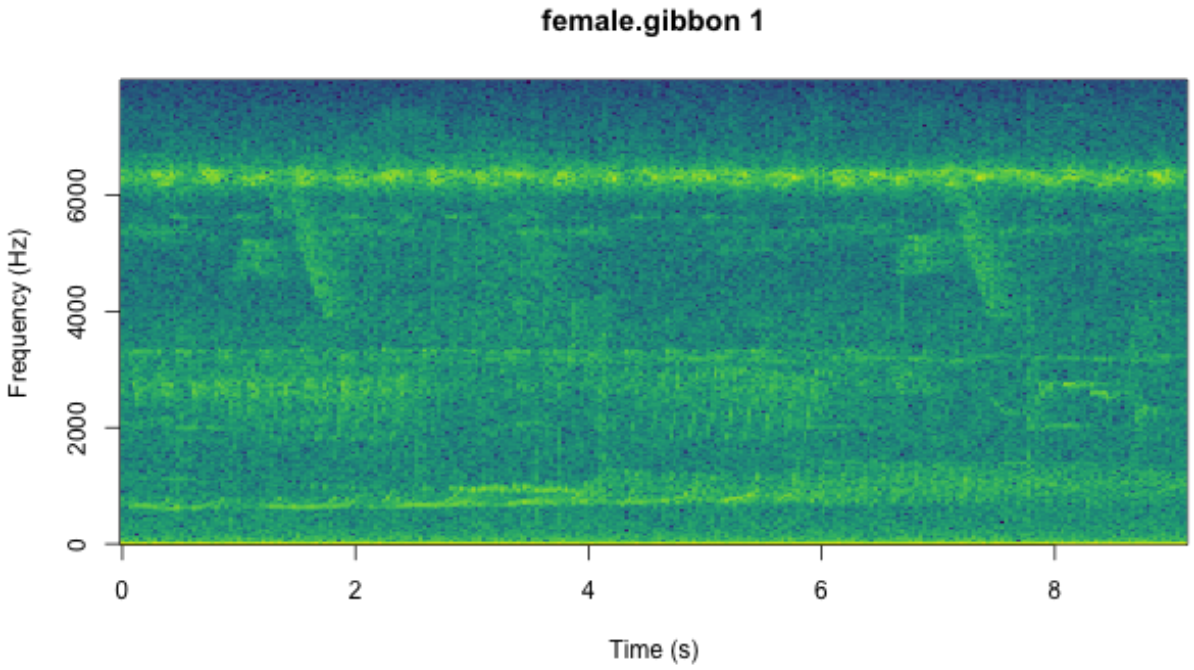
This function classifies sound events from a specified directory using user-trained machine learning algorithms.

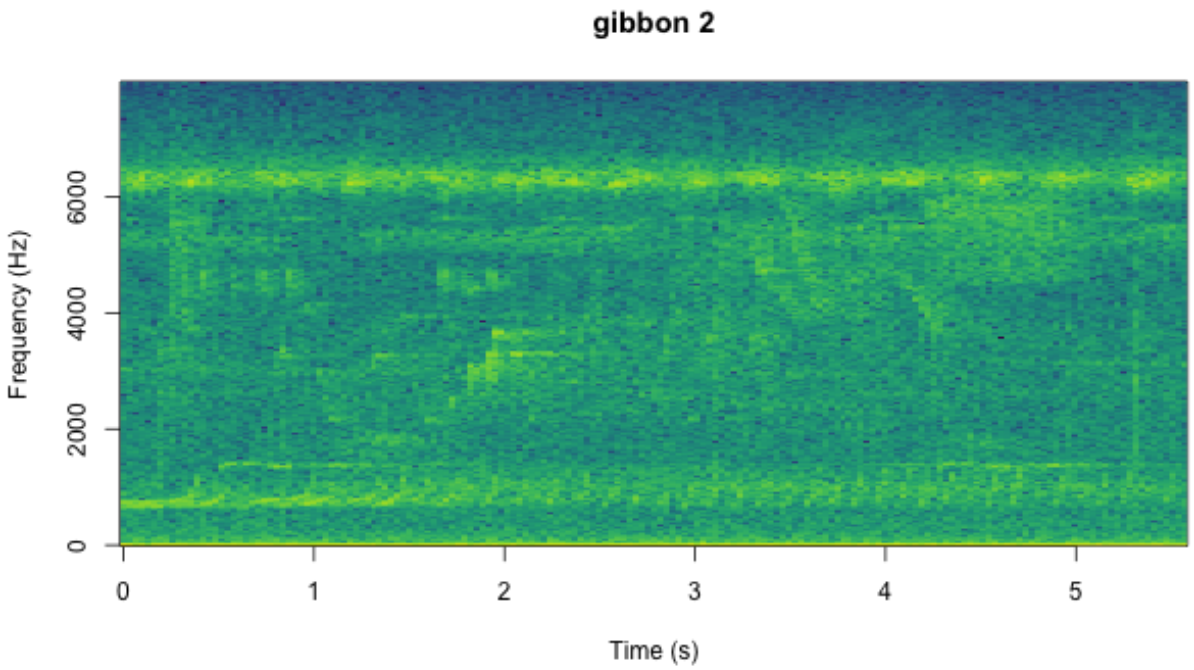
We will now load a training dataset of MFCCs that was calculated for 4337 observer-labeled sound events.

```
# MFCC settings were as follows: min.freq=400, max.freq=2000, n.window=9, n.cep=12.
data("training.MFCC.long")
```

This function will read in a set of .wav files that were created using either method of audio segmentation (GMM or SVM). It can also be used on user-labeled calls in a directory. This function requires the user to input labeled training data and gives the option to use SVM, GMM or NNET models for classification. The option is set to plot=TRUE which means that spectrograms for each sound event will be plotted along with the predicted class.

```
classification.df <- classifyGibbonR(
  feature.df = training.MFCC.long,
  model.type = "SVM",
  tune = FALSE,
  input.dir = "/Users/denasmacbook/Desktop/output.test/temp",
  plot = TRUE,
  min.freq = 400,
  max.freq = 2000,
  n.window = 9,
  n.cep = 12
)
```






```
# The function returns a dataframe which includes the file name, model prediction and probability
classification.df
```

AUTOMATED SIGNAL DETECTION

This function combines all of the previous steps into one function. STEP 1. Audio segmentation
STEP 2. Train the machine learning classifier using observer labeled data
STEP 3. Run the automated detector
STEP 4. User validation of the system

```
# Assign either a .wav file or full file path to .wav file with signal(s) of interest
# Here we will use one of the previously loaded .wav files
wav.for.detection <- wav.1
```

Create the MFCC feature training data set. NOTE: You will need to ensure the MFCC setting for the calcMFCC are the same as for the detectGibbonR function particularly n.window, n.cep, min.freq, max.freq

```
training.MFCC <- calcMFCC(
  list.wav.files = multi.class.list,
  n.window = 9,
  n.cep = 12,
  min.freq = 400,
  max.freq = 2000,
  feature.red = FALSE
)
```

```
# Check structure of dataframe
str(training.MFCC)
```

```
# Convert specific class names to more general names if desired
 training.MFCC$class <-
   plyr::revalue(training.MFCC$class,
                 c("female.gibbon" = "gibbon", "solo.gibbon" = "gibbon"))
```

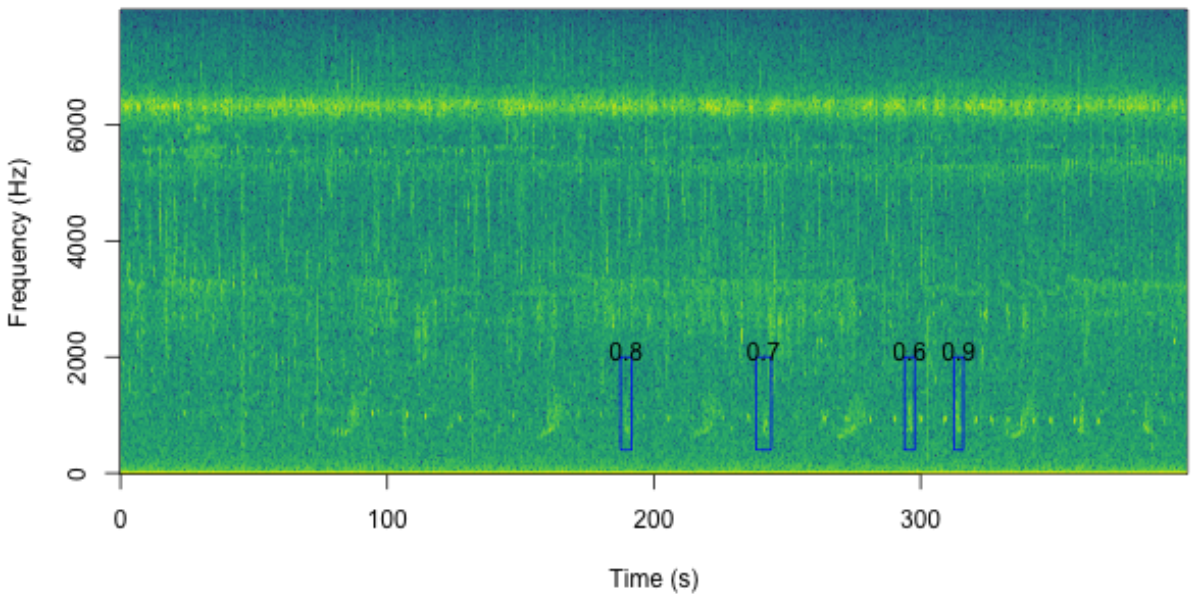
detectGibbonR FOR AUTOMATED DETECTION OVER A SINGLE SOUND FILE

```
detectiontiming <-
  detectGibbonR(
    feature.df = training.MFCC,
    model.type = "SVM",
    tune = "FALSE",
    wav.name = wav.for.detection,
    which.quant = "low.quant",
    min.freq = 0.4,
    max.freq = 2,
    low.quant.val = 0.05,
    n.windows = 9,
    min.sound.event.dur = 3,
    target.signal = "female.gibbon",
    probability.thresh = 0.55,
    output.dir = output.dir
  )
```

Visualize results.

```
# First calculate the spectrogram matrix
temp.spec <-
```

```
  signal::specgram(wav.for.detection@left,
                   Fs = wav.for.detection@samp.rate,
                   n = 512,
                   overlap = 0)

# Then plot the spectrogram
plot(
  temp.spec,
  xlab = "Time (s)",
  ylab = "Frequency (Hz)",
  col = viridis::viridis(512),
  useRaster = TRUE
)

# And add boxes for the identified sound events
for (x in 1:nrow(detectiontiming$timing.df)) {
  rect(detectiontiming$timing.df[x, 3],
       400,
       detectiontiming$timing.df[x, 4],
       2000,
       border = "blue")
  vec <-
    c(detectiontiming$timing.df[x, 3], detectiontiming$timing.df[x, 4])
  x.val <- vec[-length(vec)] + diff(vec) / 2
  text(x.val, 2100,
       labels = round(detectiontiming$timing.df[x, 5], digits = 1))
}
```



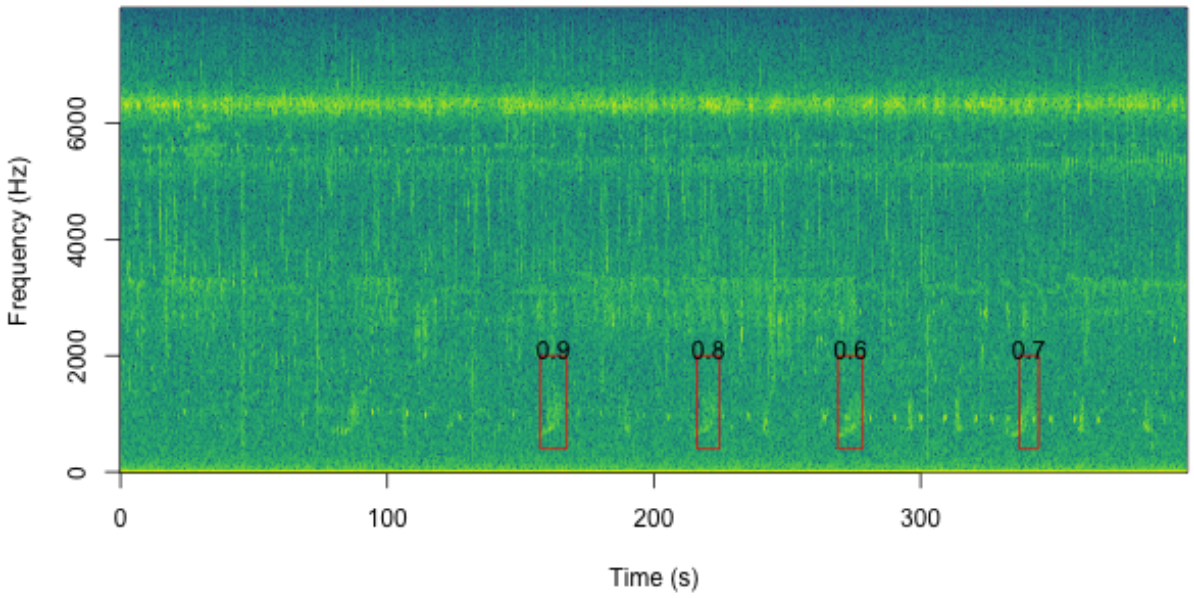
This detector is not performing very well... Which is related to the minimal amount of training data. What

happens when we train with a more extensive training dataset? NOTE: The code below will take a substantial amount of time to run.

```
data("training.MFCC.long")
# Function to detect gibbon calls from long-term recordings
detectiontiming <-
  detectGibbonR(
    feature.df = training.MFCC.long,
    tune = "FALSE",
    wav.name = wav.for.detection,
    model.type = "SVM",
    min.freq = 0.4,
    max.freq = 2,
    which.quant = "low.quant",
    n.windows = 9,
    min.sound.event.dur = 3,
    target.signal = "female.gibbon",
    probability.thresh = 0.55,
    output.dir = output.dir
  )
```

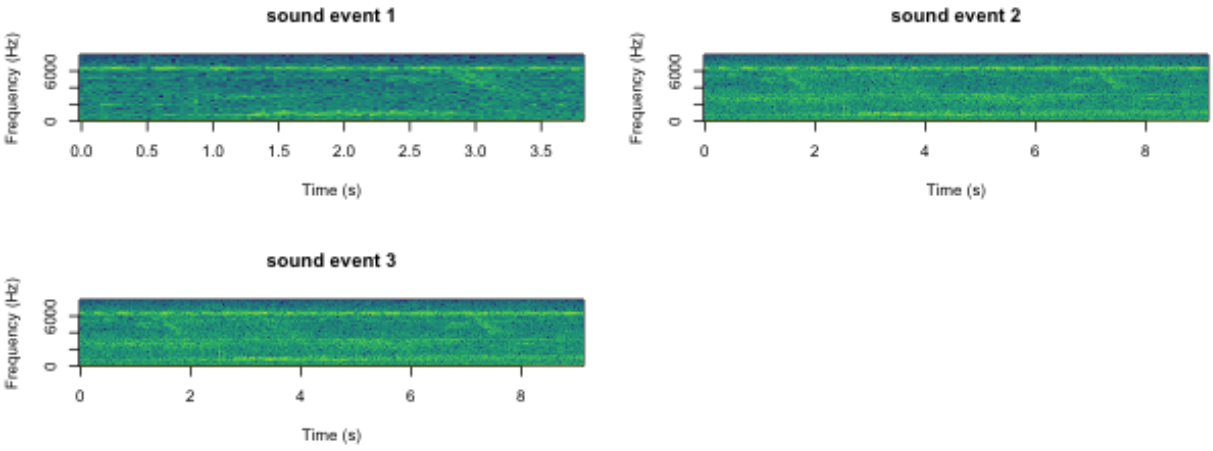
Visualize results.

```
# First calculate the spectrogram matrix
temp.spec <-
  signal::specgram(wav.for.detection@left,
                   Fs = wav.for.detection@samp.rate,
                   n = 512,
                   overlap = 0)

# Then plot the spectrogram
plot(
  temp.spec,
  xlab = "Time (s)",
  ylab = "Frequency (Hz)",
  col = viridis::viridis(512),
  useRaster = TRUE
)

# And add boxes for the identified sound events
for (x in 1:nrow(detectiontiming$timing.df)) {
  rect(detectiontiming$timing.df[x, 3],
       400,
       detectiontiming$timing.df[x, 4],
       2000,
       border = "red")
  vec <-
    c(detectiontiming$timing.df[x, 3], detectiontiming$timing.df[x, 4])
  x.val <- vec[-length(vec)] + diff(vec) / 2
  text(x.val, 2100,
       labels = round(detectiontiming$timing.df[x, 5], digits = 1))
}
```



We can then plot the individual sound events from the specified output directory.

```
plotSoundevents(input.dir = output.dir, n.soundevents = 3,
                nrow = 3,
                ncol = 2)
```



batchDetectGibbonR FOR AUTOMATED DETECTION OVER MULTIPLE SOUND FILES

```
detect.df <- batchDetectGibbonR(
  input = listBorneoSounds,
  feature.df = training.MFCC,
  model.type = "SVM",
  min.freq = 0.4,
  max.freq = 2,
  which.quant = "low.quant",
  n.windows = 9,
  min.sound.event.dur = 3,
  wav.output = TRUE,
  target.signal = "female.gibbon",
  probability.thresh = 0.55,
  output.dir = output.dir
)
```

CREATING A CALL DENSITY PLOT

This requires the user to input a .csv file with latitude and longitude

```
data("temp.detect.df")
data("gps.df")

calldensityPlot(gps.df = gps.df,
                    calltiming.df = temp.detect.df)
```