Tutorial For SCAMPI

2024-09-24

This tutorial provides an example for implementing SCAMPI model (Chen, Groh, and Tokdar 2024).

Install the package

To implement the SCAMPI model, we'll use the `ratejuggling` function from the `neuromplex` package. The most recent version of the `neuromplex` package is `neuromplex_1.0-6`. To install it, download the `neuromplex_1.0-6.tar.gz` file and install the package from the local file.

If you have the `neuromplex_1.0-6.tar.gz` ready, you can install it using the following code:

```r
install.packages("path/to/your/package.tar.gz", repos = NULL, type = "source")
```

Here

- `path/to/your/package.tar.gz`: Replace this with the actual path to your `.tar.gz` file.
- `repos = NULL`: install the package from a local file instead of a repository.
- `type = "source"`: install from a source package.

Load the packakge

Once you have the latest version of `neuromplex`, load it to the R workspace.

```r
library(neuromplex)
```

Example for implementing SCAMPI model

Introduction of `ratejuggling()`

To implement the SCAMPI model exactly as described by (Chen, Groh, and Tokdar 2024), you can use the `ratejuggling()` function from the `neuromplex` package. The function's default arguments are sufficient for most cases, so you'll only need to provide `xA`, `xB`, and `xAB`.

Main arguments of `ratejuggling()`:

1. **Data-related arguments:**
   - `xA`, `xB`, `xAB`: These arguments take the spike counts for conditions `A`, `B`, and `AB`, respectively.
   - `remove.zeros`: Controls whether trials with zero spikes are removed. The default is `remove.zeros = FALSE`.
2. **Prior-related arguments:**

    - `gamma.pars = c(0.5, 2e-10)`: This suggests that we do not have strong prior information about the firing rate.
    - `beta.pars = c(0.5, 0.5)`: This indicates a neuron has an equal prior probability of encoding condition `A` or `B` when both stimuli are present, without strong prior belief.

3. **SCAMPI model-related arguments:**
    - `method.screen = variance`: Uses a Poisson variance test to check whether a Poisson distribution fits the sample counts. In (Chen, Groh, and Tokdar 2024), the Fano factor is used to screen triplets.
    - `mix.method = "PRML"`: Applies the predictive recursive algorithm introduced in (Chen, Groh, and Tokdar 2024) to estimate marginal likelihood for each model.
    - `fixed.method = "jeffreys"`: Uses Jeffreys' prior for the Fixed Poisson hypothesis.

Output of `ratejuggling()`:

- **`separation.logBF`**: The logarithm of the Bayes factor testing whether two single stimulus distributions are different.
- **`post.prob`**: Posterior probabilities for four hypotheses: slow juggling, fast juggling, fixed Poisson, and overreaching. These probabilities can be further processed to categorize the fixed Poisson model into four groups: preferred, non-preferred, middle, and outside.
- **`pois.pvalue`**: The minimum of two p-values that check for the Poisson distribution of each single stimulus distribution, calculated based on the `method.screen` specified.
- **`sample.sizes`**: The sample sizes for the `A`, `B`, and `AB` conditions.

Example for implementing SCAMPI model in a raw dataset

Here presents an example code for embedding `ratejuggling` to our pipeline. Notice the dataset needs save in a format as we specified in the pipeline.

```r
library(tidyverse)
library(neuromplex)
library(weights)
library(gtools)

## helper functions for readability
# short helper functions for

# string manipulation to get cell_id and trialparams file names
get_trial_params_name <-function(spiketimes_name) {
  foo<-str_locate(spiketimes_name,"_cell")
  cell_id<-substr(spiketimes_name,1,foo[1]-1)
  trial_params_name<-str_c(cell_id,"_trialparams.csv",collapse=NULL)
  return(trial_params_name)
}

get_spike_counts<-function(trial_list,spiketimes,starttime,stoptime){
  spike_counts<-vector(mode = "numeric",length=length(trial_list[,1])) # initialize a numeric vector

  for (n in c(1:length(trial_list[,1]))){ # loop across individual trials
    this_trial<-trial_list[n,1]

    #get the spikes if 1) correct trial number; 2)spiketime>starttime; 3) spiketime<stoptime
    #this_spikes<-spiketimes[spiketimes[,1]==this_trial&spiketimes[,2]>starttime&spiketimes[,2]<=stopti
    this_spikes<-spiketimes[spiketimes[,1]==this_trial,2] #trial match?
    spikes_to_use<-this_spikes[this_spikes>starttime & this_spikes<=stoptime] #time window?
    spike_counts[n]<-length(spikes_to_use)
    #count is the number of entries after the above screens
  }
  return(spike_counts)
}

get_spike_counts_bins<-function(trial_list,spiketimes,starttime,stoptime,binwidth){
  num_bins<-floor((stoptime-starttime)/binwidth) #
  #spike_bin_counts<-array(numeric(),c(num_bins,length(trial_list[,1]),0)) # initialize a numeric array
  spike_bin_counts<-matrix(0L,nrow=num_bins,ncol=length(trial_list[,1]))
  curr_start<-starttime
  curr_stop<-curr_start+binwidth

  for (this_bin in c(1:num_bins)){ # outer loop is time bins
    for (n in c(1:length(trial_list[,1]))){ # inner loop across individual trials
      this_trial<-trial_list[n,1]

      #get the spikes if 1) correct trial number; 2)spiketime>starttime; 3) spiketime<stoptime
      #this_spikes<-spiketimes[spiketimes[,1]==this_trial&spiketimes[,2]>starttime&spiketimes[,2]<=stop
      this_spikes<-spiketimes[spiketimes[,1]==this_trial,2] #trial match?
      spikes_to_use<-this_spikes[this_spikes>curr_start & this_spikes<=curr_stop] #time window?
      spike_bin_counts[this_bin,n]<-length(spikes_to_use) #count is the number of entries after the abo
    }
    curr_start<-curr_stop # move the bin
    curr_stop<-curr_stop+binwidth
  }
  return(spike_bin_counts)
}


## settings from Jenni's pipline

run_dapp_flag<-FALSE #EDIT set to FALSE to run faster and skip the DAPP; Use FALSE for all Cohen2 datas
run_prml_flag<-TRUE #EDIT set to FALSE to run faster and skip the PRML
save_dapp_outputs<-FALSE #EDIT set to FALSE to take up less disk space;

### PATHS to code, data, and output###
wheres_my_code="" #EDIT
wheres_my_data<-"" #EDIT
wheres_my_output<-""
wheres_my_top_level_output<-""
my_output_file_name<-"" #EDIT

my_search_string<-"_spiketimes\\.csv" #EDIT -

desired_avg_bin_height<-10 # this is for the spike count histograms, use 15 if you have a lot

my_A_expression<-"COND==condA & REWARD == 1"
my_B_expression<-"COND==condB & REWARD == 1"
my_AB_expression<-"COND==condAB & REWARD == 1"
####  EDIT RESPONSE PERIOD DURATION  #################################
```

```r
# SOFF is shorthand for "stim off" - is not necessarily stim off though - think of it as spikes off
min_SOFF_flag<-FALSE #set to be TRUE if you want to find the SOFF from the data
my_min_SOFF<-450 #only used if min_SOFF_flag is FALSE
my_min_SON<-50
#zero for most datasets, can be a negative value for saccade aligned analysis
my_binwidth<-50 # for the DAPP

base_starttime<--500 #set both start and stop to 0 if you don't want/have baseline spikes
base_stoptime<-0
base_dur<-base_stoptime-base_starttime
# This does not have to match the duration of the stimulus response period;
# normalization to make it proportional is done later
###  No more #EDITs after this point, if all goes well.############################################

####  Go to the code directory
setwd(wheres_my_data)
outfile<-paste(wheres_my_output,my_output_file_name,sep="/") #combine path and file names

list_of_files<-list.files(path = wheres_my_data, pattern = my_search_string, all.files = FALSE,
                          full.names = FALSE, recursive = FALSE,
                          ignore.case = FALSE, include.dirs = FALSE, no.. = FALSE)
output_regardless_of_sepBF<-FALSE
read_custom_triplets<-FALSE

try({for (file_n in c(1:length(list_of_files))){
  #for (file_n in c(1:2)){ # if you need to run a short version
  # give some periodic status updates

  this_neuron_spiketimes_name<-list_of_files[[file_n]]
  this_neuron_trialparams_name<-get_trial_params_name(this_neuron_spiketimes_name)

  tp_data<-read.csv(this_neuron_trialparams_name)    # load the trial params data
  tp_data$COND<-gsub(" ","",tp_data$COND)
  #strip any whitespace in the COND column & remake it as a factor
  st_data<-read.csv(this_neuron_spiketimes_name,header = F) # load the spiketimes data
  # get a list of all the "Double" conditions
  list_of_double_conds<-unique(tp_data$COND[tp_data$SIGNAL=="Double"])

  ## Now loop across the list of double conditions
  ## and parse out the matching A and B conditions
  for (n in c(1:length(list_of_double_conds))){
      condAB=list_of_double_conds[n]
      foo<-str_split(list_of_double_conds[n],'\\+',simplify = T) # find the plus sign; the \\ is becaus
      condA<-foo[1,1]
      #the A and B conditions are the portions of the strings
      #before and after the plus sign
      condB<-foo[1,2]

    # go get the trial lists for A, B, and AB conditions;
    # note the extra selection criteria not yet implemented
    Atrials<-subset(tp_data,eval(parse(text=my_A_expression)))
    # my_A_expression etc. set in the custom portion
    Btrials<-subset(tp_data,eval(parse(text=my_B_expression)))
    ABtrials<-subset(tp_data,eval(parse(text=my_AB_expression)))

    # do we meet the 5-5-5 rule, i.e. min 5 trials for each condition?
    # if not, skip and go on to the next
    if (min(c(length(Atrials[,1]),length(Btrials[,1]),length(ABtrials[,1])))<5){next}

    #Next steps: get the Acounts, Bcounts, and ABcounts from 0 to SOFF
    #Need to check for minimum SOFF in a triplet
    if (min_SOFF_flag==TRUE) {
      minSOFF<-min(c(Atrials[,"SOFF"],Btrials[,"SOFF"],ABtrials[,"SOFF"]))
      # this line finds the min SOFF value across conditions,
      #ensures spike counting is appropriate and same for all conditions in the triplet
    } else { minSOFF<-my_min_SOFF}
    # use the value specified by the user in *custom_call_to_wrapper_runner.R

    Acounts<-get_spike_counts(trial_list=Atrials,spiketimes=st_data,starttime=my_min_SON,stoptime=minSOF
    Bcounts<-get_spike_counts(trial_list=Btrials,spiketimes=st_data,starttime=my_min_SON,stoptime=minSOF
    ABcounts<-get_spike_counts(trial_list=ABtrials,spiketimes=st_data,starttime=my_min_SON,stoptime=minS
    res_sum=c(mean(Acounts,na.rm=T),mean(Bcounts,na.rm=T),mean(ABcounts,na.rm=T),
              var(Acounts,na.rm = T)/mean(Acounts,na.rm=T),
              var(Bcounts,na.rm = T)/mean(Bcounts,na.rm=T),
              var(ABcounts,na.rm = T)/mean(ABcounts,na.rm=T))
    ### You only need to provide `xA`, `xB`, and `xAB`;
    ###the remaining arguments can be left at their default values.
      new_whole_trial_results<-ratejuggling(Acounts, Bcounts, ABcounts)
      all_res=list(res_sum,new_whole_trial_results)
      cat(this_neuron_spiketimes_name,unique(ABtrials$COND), unlist(all_res), "\n", file = outfile, appe
      } # end of loop across triplets
}}) # top of the loop across files
```

Post process the raw output from `ratejuggling` and visualization

```
### read SCAMPI results

dat.pth="path/to/your/SCAMPIresults/"

raw=read.table(file = paste0(dat.pth,"filename.txt"),header = FALSE,sep = " ",stringsAsFactors = FALSE,r
  select(-V20) # if you results also have an empty column, remove it
names(raw)=c("file","cond","meanA","meanB","meanAB",
             "fanoA","fanoB","fanoAB","SepBF",
             "PrSlowJug","PrFastJug","PrFix","PrOvReach",
             "PvalA","PvalB","PvalAB",
             "nA", "nB", "nAB")

### Post processing:
# function to add `winning model` and `winning probability` to the original dataframe,
# and subcategorize Fixed Poisson triplets.
get.res <- function(fit){
  model.names=c("SlowJug","FastJug","Fixed","Overreaching")
  pModel=fit %>% select(PrSlowJug:PrOvReach)
  WinModel <- apply(pModel, 1, function(z) model.names[which.max(z)])
  WinPr <- apply(pModel,1,max)
  res=fit %>%
    mutate(WinModel=WinModel,
           WinPr=WinPr)
  typ <- with(res,WinModel)
  prb <- with(res,WinPr)
  fixed <- (typ == 'Fixed')
  if(any(fixed)){
    subtyp.fixed <- with(res[fixed,], cbind(PrSlowJug,PrFastJug,PrOvReach))
    prb[fixed] <- prb[fixed]*apply(subtyp.fixed,1,max)/apply(subtyp.fixed,1,sum)
    typ[fixed] <- c('Single','FxdMid','FxdOut')[apply(subtyp.fixed,1,which.max)]
  }
  single <- (typ == 'Single')
  if(any(single)){
    prefA <- with(res[single,],meanA > meanB)
    meanPref <- with(res[single,],pmax(meanA,meanB))
    meanNonp <- with(res[single,],pmin(meanA,meanB))
    subtype.single <- res$meanAB[single]*log(meanPref/meanNonp) > (meanPref - meanNonp)
    typ[single] <- c('FxdNon','FxdPrf')[subtype.single+1]
  }
  res$Type <- factor(
    typ,
    levels=c('FastJug','SlowJug','Overreaching',
             'FxdPrf','FxdNon','FxdMid','FxdOut'),
    labels=c('FastJug','SlowJug','OvReach',
             'FxdPrf','FxdNon','FxdMid','FxdOut'))
  res$Prob <- prb
  return(res)
}
raw = get.res(na.omit(raw))

### Filtering and Visualization


library(ggstats)
library(ggplot2)
cbPalette <- c("#999999", "#E69F00", "#56B4E9", "#009E73", "#F0E442", "#0072B2", "#D55E00", "#CC79A7")

minsize <- 5

plotSCAMPI=raw %>%
  filter(SepBF>=3,
         nA >= minsize, nB >= minsize, nAB >= minsize,
         fanoA<3, fanoB<3) %>%
  mutate(Confidence=factor(cut(WinPr,c(0,.5,.75,.95,1)))) %>%
  ggplot(., aes(x=Type,y=after_stat(prop),by = 1)) +
  geom_bar(aes(fill=Confidence),color='black',stat = "prop") +
  scale_y_continuous(labels = scales::percent,
                     limits = c(0,1)) +
  geom_text(
    aes(
      label = scales::percent(after_stat(prop), accuracy = .1)),
    stat = "prop",
    position=position_stack(),
```

```r
    vjust = -0.5,
    size=3) +
  scale_fill_manual(values=cbPalette,drop=FALSE) +
  scale_x_discrete(drop=FALSE)+
  theme_bw() +
  theme(axis.text.x = element_text(angle = 45, vjust = 0.5, hjust=0.5))
plotSCAMPI
# save the plot
ggsave(plotSCAMPI,filename = paste0(pth,"name_of_plot.jpeg"),height = 5,width = 9)
```

References

Chen, Yunran, Jennifer M Groh, and Surya T Tokdar. 2024. "Spike Count Analysis for MultiPlexing Inference (SCAMPI)." *bioRxiv*, 2024–09.