kernel_estimator_selection_bootstrapping

1 - Cross-Validation For Kernel Estimators

Overview

Given independent and identically distributed interspike intervals $X_1, \ldots, X_n$, let us look at the kernel estimators of an underlying density f. We choose $K(x) = \frac{\exp(-x^2/2)}{\sqrt{2\pi}}$. The kernel estimator with bandwidth h is then defined as:

$$\hat{f}_h(u) = \frac{1}{nh} \sum_{i=1}^{n} K(\frac{u - X_i}{h})$$

A – Let's implement a function that computes, for a given vector u and a given window h, the value $\hat{f}_h(.)$ for each coordinate of the vector u. Then let's simulate exponential variables and apply the kernel estimator with different values for h.

B – Choosing h can be done with cross-validation: Let's rely on the least-square contrast:

$$C(g) = -\frac{2}{n} \sum_{i=1}^{n} g(X_i) + \int g(x)^2 dx$$

Such that $\mathbb{E}[C(g)]$ is minimal when $f = g$ where g is a candidate density. Let's demonstrate that state.

C – Using the function `integrate` , let's compute the integral of the kernel estimator f to the square and implement a function that computes the least square contrast on a different sample than the one previously simulated.

D – Let's implement the hold-out estimator: The data is cut in half at random, half to be used for estimation, and the other half for the selection of the bandwidth parameter h. Then let's plot the corresponding estimator.

E – Let's compare IC_1 and IC_2 for various N sizes, then explain why IC_2 cannot converge to IC_1 when N goes towards infinity (even if it is a pretty good approximation).

F – Let's apply the previous bootstrap heuristic on the STAR-package data to show that there is a clear difference in neuron 1 (experiment `citronellal`) between the period "before the puff", "during the puff", and "after the puff" (See STAR documentation). The level of the confidence interval will be updated to account for Bonferroni correction.

Step A

Let's declare the functions $K(.)$ and $\hat{f}_h(.)$ under the name `kernel` and `kernel_estimator` espectively:

```
kernel <- function(x) {
  # Declares of the Gaussian kernel function K(x)
  exp(-1*x^2/2)/sqrt(2*pi)
}

kernel_estimator <- function(ISI, u, h) {
  # Declares of the kernel estimator
  # note: looks like a DFT with a sliding window
  #
  # Formats the ISI as a matrix and retrieves its length
  ISI = as.matrix(ISI); n = length(ISI)
  # Precomputes the rescale factor of the kernel estimator
  rescale_factor = 1/(n*h)
  # Computes the estimator for each u-coordinate
  f_hat = c()
  for (index in 1:length(u)) {
    kerneling = sum(apply(ISI, 1, function(x){kernel((u[index]-x)/h)}))
    f_hat = cbind(f_hat, c(rescale_factor * kerneling))
  }
  # Returns the estimator
  f_hat
}
```

Now, let's simulate a set of interspike intervals ($n = 100$) via the exponential distribution using the default rate parameter $\lambda = 1$. Then let's run the kernel estimator with a preselected vector $u = (0.0, 0.1, \ldots, 10.0)^T$ over the set of h-bandwidth parameters

$\{0.1, \ldots, 0.5\}$.

```r
# Declares parameters
n = 100
u_stepsize = 0.01
u = seq(0, 9.99, u_stepsize)
h = as.matrix(seq(0.05, 0.5, 0.05))

# Simulates interspike intervals
ISI = rexp(n, rate=1)

# Computes the kernel estimator for each h
estimators = apply(h, 1, function(x) {kernel_estimator(ISI, u, x)})

# Declares the end data.frame holding the estimators
names_col = apply(h, 1, function(x){paste("h_",x,sep="")})
estimators = as.data.frame(estimators, row.names=u)
estimators = setNames(estimators, names_col) # naming cols within as.d.f fails for me
```

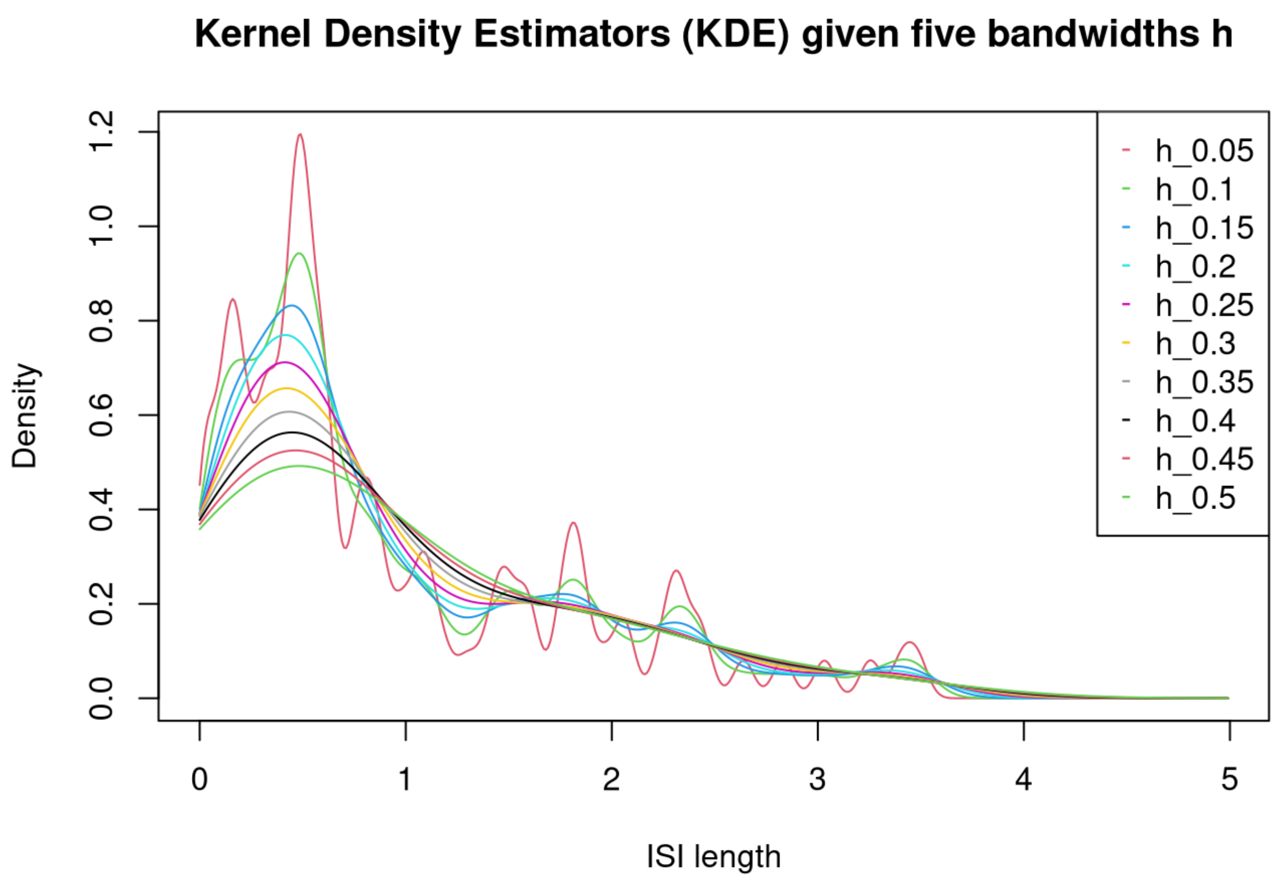
Let's display the resulting estimators over the preselected range of h bandwidths, given the vector u and the randomly generated interspike intervals.

```r
head(estimators)
```

```
##         h_0.05     h_0.1    h_0.15     h_0.2    h_0.25     h_0.3    h_0.35
## 0    0.4518733 0.4015419 0.3919459 0.3882558 0.3892401 0.3888562 0.3849358
## 0.01 0.4997358 0.4304056 0.4113868 0.4032319 0.4015249 0.3990093 0.3933208
## 0.02 0.5393312 0.4588048 0.4306412 0.4181741 0.4137991 0.4091355 0.4016642
## 0.03 0.5699738 0.4864960 0.4496279 0.4330512 0.4260438 0.4192217 0.4099570
## 0.04 0.5923997 0.5132490 0.4682699 0.4478326 0.4382403 0.4292549 0.4181899
## 0.05 0.6086639 0.5388504 0.4864957 0.4624893 0.4503697 0.4392217 0.4263536
##          h_0.4    h_0.45     h_0.5
## 0    0.3778478 0.3685993 0.3581298
## 0.01 0.3847900 0.3743814 0.3629854
## 0.02 0.3916840 0.3801145 0.3677943
## 0.03 0.3985233 0.3857939 0.3725534
## 0.04 0.4053015 0.3914151 0.3772593
## 0.05 0.4120121 0.3969735 0.3819088
```

Now, let's visualize the resulting kernel densities, truncated to the first half length of the range/vector u:

```r
plot_kde <- function(kernel_estimates, range_to_plot=c(1:500)){
  # Plots the KDE collected in a given data.frame
  plot(u[range_to_plot], kernel_estimates[range_to_plot,1],
       type="l", col=2, ylab="Density", xlab="ISI length",
       main="Kernel Density Estimators (KDE) given five bandwidths h")
  for (cl in 2:dim(kernel_estimates)[2]) {
    lines(u[range_to_plot], kernel_estimates[range_to_plot,cl],
          type="l", col=cl+1)
  }
  legend("topright", legend=colnames(kernel_estimates),
         col=c(2:(dim(kernel_estimates)[2]+1)), pch="-")
}

plot_kde(estimators)
```



Step B

Let's set the least-square contrast for density C such that:

$$\forall i \in \{1, \dots, n\},\ X_i \sim f,\ \text{IID}$$
$$g,\ \text{a candidate density}$$
$$C(g) = -\frac{2}{n}\sum_{i=1}^{n} g(X_i) + \int g(x)^2 dx$$

As such, the expectation of C with g a candidate density is:

$$\begin{aligned}
\mathbb{E}_{\forall i \in \{1,\dots,n\},\, X_i \sim f}\big[C(g)\big] &= \mathbb{E}\big[-\frac{2}{n}\sum_{i=1}^{n} g(X_i) + \int g(x)^2 dx\big] \\
&= -\frac{2}{n}\sum_{i=1}^{n}\int g(x)f(x)dx + \int g(x)^2 dx \\
&= -2\int g(x)f(x)dx + \int g(x)^2 dx \\
&= \int (g(x) - f(x))^2 dx - \int f(x)^2 dx
\end{aligned}$$

Given that $\int f(x)^2 dx$ is a constant, $\mathbb{E}_{\forall i \in \{1,\dots,n\},\, X_i \sim f}\big[C(g)\big]$ is minimal when $\int (g(x)f(x))^2 dx$ is minimal. As such, it is demonstrable that:

$$\int (g(x) - f(x))^2 dx \geq 0$$
$$\int (g(x) - f(x))^2 dx = 0 \text{ if and only if } \forall x,\ g(x) - f(x) = 0$$

It can be concluded that $\mathbb{E}[C(g)]$ is minimal when the candidate density g is equal to the density f given $\forall i \in \{1, \dots, n\},\ X_i \sim f,\ \text{IID}.$

Step C

Let's build the function implementing $C(f)$ such that:

$$C(f) = -\frac{2}{n}\sum_{i=1}^{n} f(X_i) + \int f(x)^2 dx$$

The integral part can be computed by reusing the previously declared function `kernel_estimator`. Let's also pre-emptively create a contrast function that can handle a train and a validation/test sets for the next steps D and E.

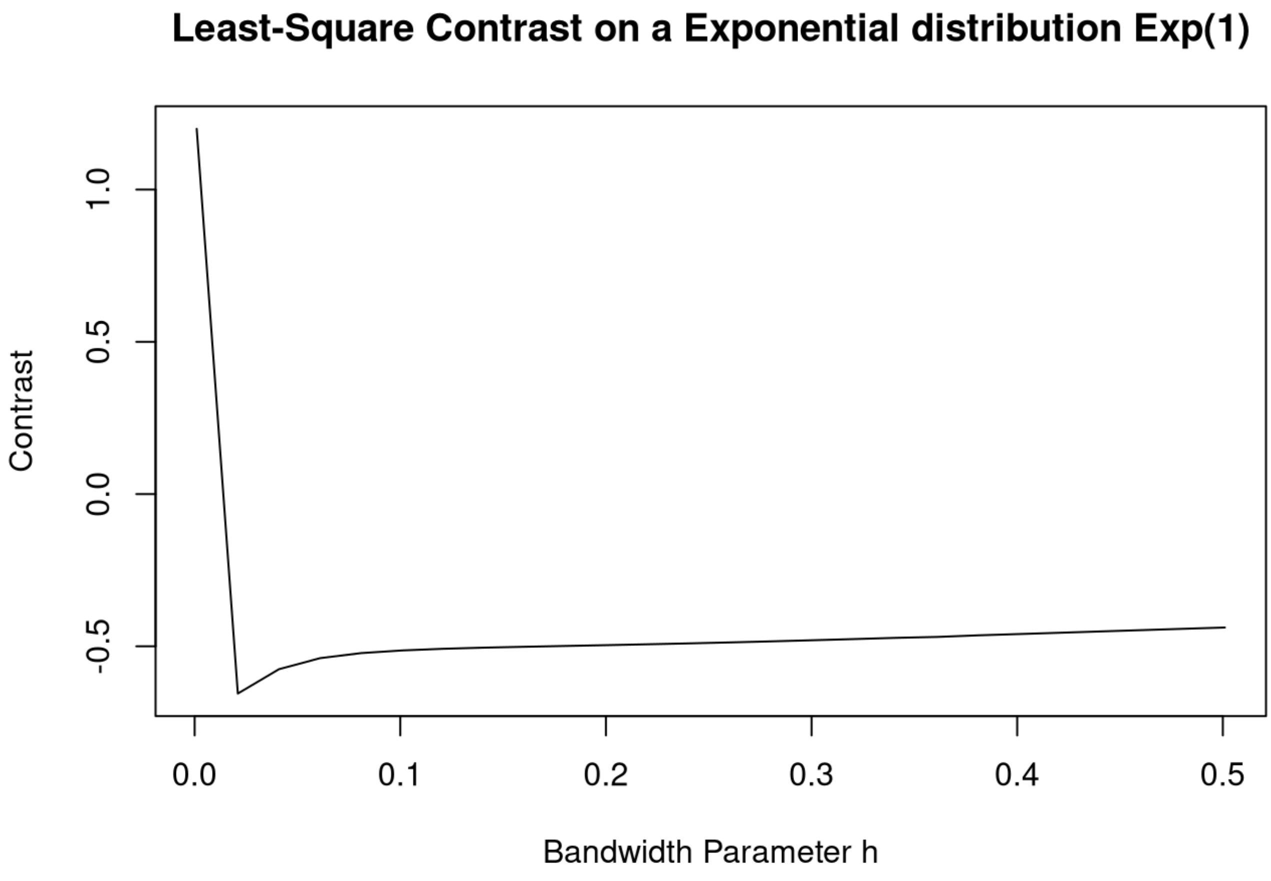
```r
integrate_kde <- function(kde) {
  # Integrates a kde function over its support (here R_+)
  integrate(Vectorize(kde),
            lower=0, upper = Inf,
            subdivisions=5000)
}

contrast_least_squares <- function(ISI_train, u, u_stepsize, h, ISI_test=NULL){
  # Declares the least-square contrast function
  #
  # Formats the ISI as a matrix and retrieves its length
  ISI_train = as.matrix(ISI_train)
  if (!is.null(ISI_test)) {
    ISI_test = as.matrix(ISI_test); n = length(ISI_test)
  } else {
    ISI_test = as.matrix(ISI_train); n = length(ISI_train)
  }
  # Computes the kernel estimators and the corresponding KDE
  estimators = kernel_estimator(ISI_train, u, h)
  # Declares useful functions to compute the densities
  na_check <- function(x) {if (is.na(x) || is.null(x)) {-Inf} else {x}}
  support_check <- function(x) {if (x < min(u) || x > max(u)) {T} else {F}}
  kde <- function(x) {
    if (support_check(na_check(x))) {
      0
    } else {
      x = as.vector(estimators)[round(x/u_stepsize)]
      if (length(x)==0 || is.na(x)) {0} else {x}
    }
  }
  kde_squared <- function(x) {kde(x)^2}
  # Computes the relative likelihood of each X given the computed
  # Kernel Density Estimation
  densities = apply(ISI_test, 1, kde)
  # Computes the corresponding integral of the kernel estimator
  integration = integrate_kde(kde_squared)
  #cat(paste("Integration result: ", integration,"\n"))
  # Computes and return the contrast
  contrast = -1*2/n*sum(unlist(densities), na.rm=T) + integration$value
  contrast
}
```

Now, let's generate a new sample, drawn from the same exponential distribution as previously stated. Let's also update the support for the bandwidth parameters h so as to have a finer parameter grid. The sample size is increased from 100 to 1000.

```r
# Updates parameters
h = as.matrix(seq(0.01, 0.5, 0.01))
n = 100

# Simulates interspike intervals
ISI = rexp(n, rate=1)
```

Reusing the previous parameters, the contrast for the newly simulated ISI can be computed:

```r
# Declares parameter
u_stepsize = 0.01
u = seq(0., 5., u_stepsize)
h = as.matrix(seq(0.001, 0.51, 0.02))

# Computes the resulting contrasts
contrasts = apply(h,1,function(x) {
  contrast_least_squares(ISI, u, u_stepsize, x)
})

# Plots the given contrasts for the set of bandwidths h
plot(h, contrasts, type="l",
     xlab="Bandwidth Parameter h", ylab="Contrast",
     main="Least-Square Contrast on a Exponential distribution Exp(1)")
```



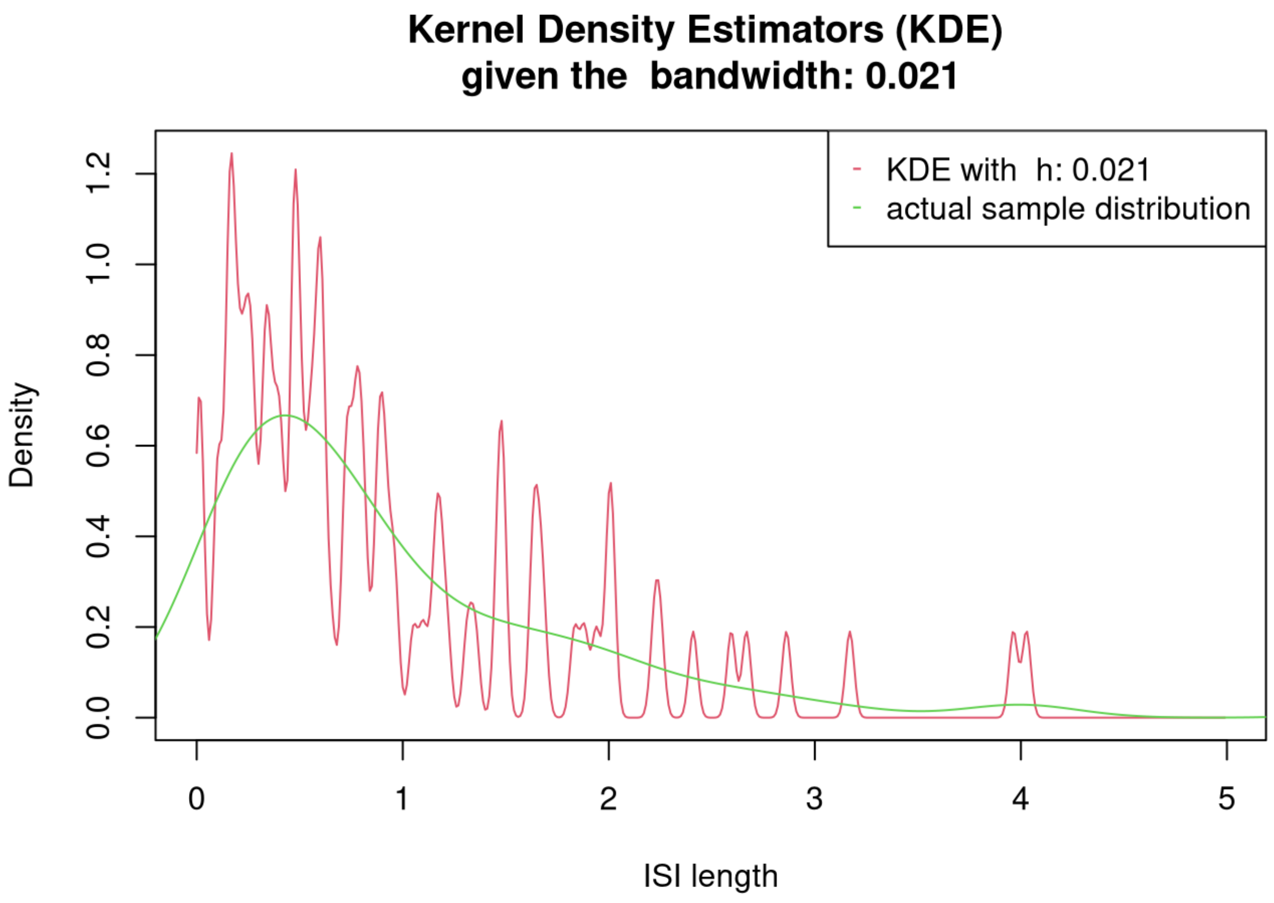
The lowest contrast was achieved on the new simulation of the exponential distribution $\mathcal{E}(1)$ with the bandwidth parameter h set to:

```r
best_h = h[which(contrasts==min(contrasts))]
best_h
```

```
## [1] 0.021
```

With this bandwidth value, let's compare the kernel estimator's resulting distribution against the actual sample distribution.

```
# Declares the plotting function
plot_kde_vs_sample_dist <- function(estimator, ISI, best_h, val_method=""){
    # Plots the KDE collected in a given data.frame
  range_to_plot = 1:500
  plot(u[range_to_plot], estimator[range_to_plot],
         type="l", col=2, ylab="Density", xlab="ISI length",
         main=paste("Kernel Density Estimators (KDE)",
                    "\ngiven the", val_method, "bandwidth:",best_h))
  lines(density(ISI), type="l", col=3)
  legend("topright", legend=c(paste("KDE with", val_method, "h:",best_h),
                              "actual sample distribution"), col=c(2, 3),
         pch="-")
}

# Computes the kernel estimates again with the selected h
estimator = kernel_estimator(ISI, u, best_h)

# Plots the distribution
plot_kde_vs_sample_dist(estimator, ISI, best_h)
```



It is evidenced that the resulting bandwidth parameter h does not specifically help fit the actual distribution. A better way to determine which h is the best is needed. This will be looked at in the next step with hold-out and cross-validation.

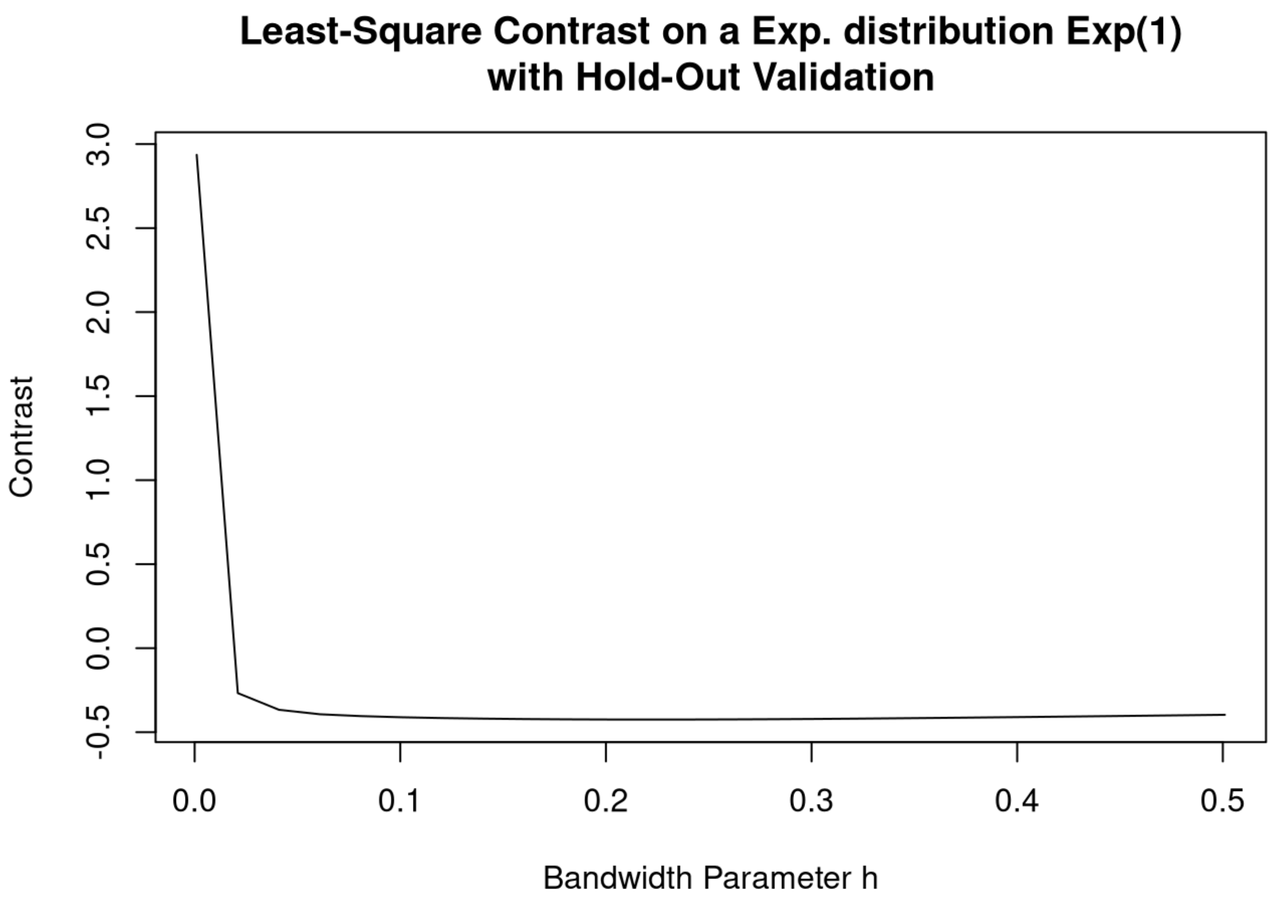
Step D

In order to perform hold-out validation, let’s start with implementing the data processing pipeline for the hold-out estimator.

```
dataset_split_holdout <- function(dataset) {
  # Implements a hold-out dataset splitting function. It
  # randomly split the input data into two equal-length subsets.
  n = length(dataset)
  split_point = round(n/2)
  scrambled_data = sample(dataset)
  return(list(
    "train"=scrambled_data[1:split_point],
    "test"=scrambled_data[(1+split_point):n])
  )
}

sets = dataset_split_holdout(ISI)
```

With both training and validation/test sets, let's run the same process as before:

```
# Computes the resulting contrasts
contrasts = apply(h,1,function(x) {
  contrast_least_squares(sets$train, u, u_stepsize, x, sets$test)
})

# Plots the given contrasts for the set of bandwidths h
plot(h, contrasts, type="l",
     xlab="Bandwidth Parameter h", ylab="Contrast",
     main="Least-Square Contrast on a Exp. distribution Exp(1)\nwith Hold-Out Validation")
```



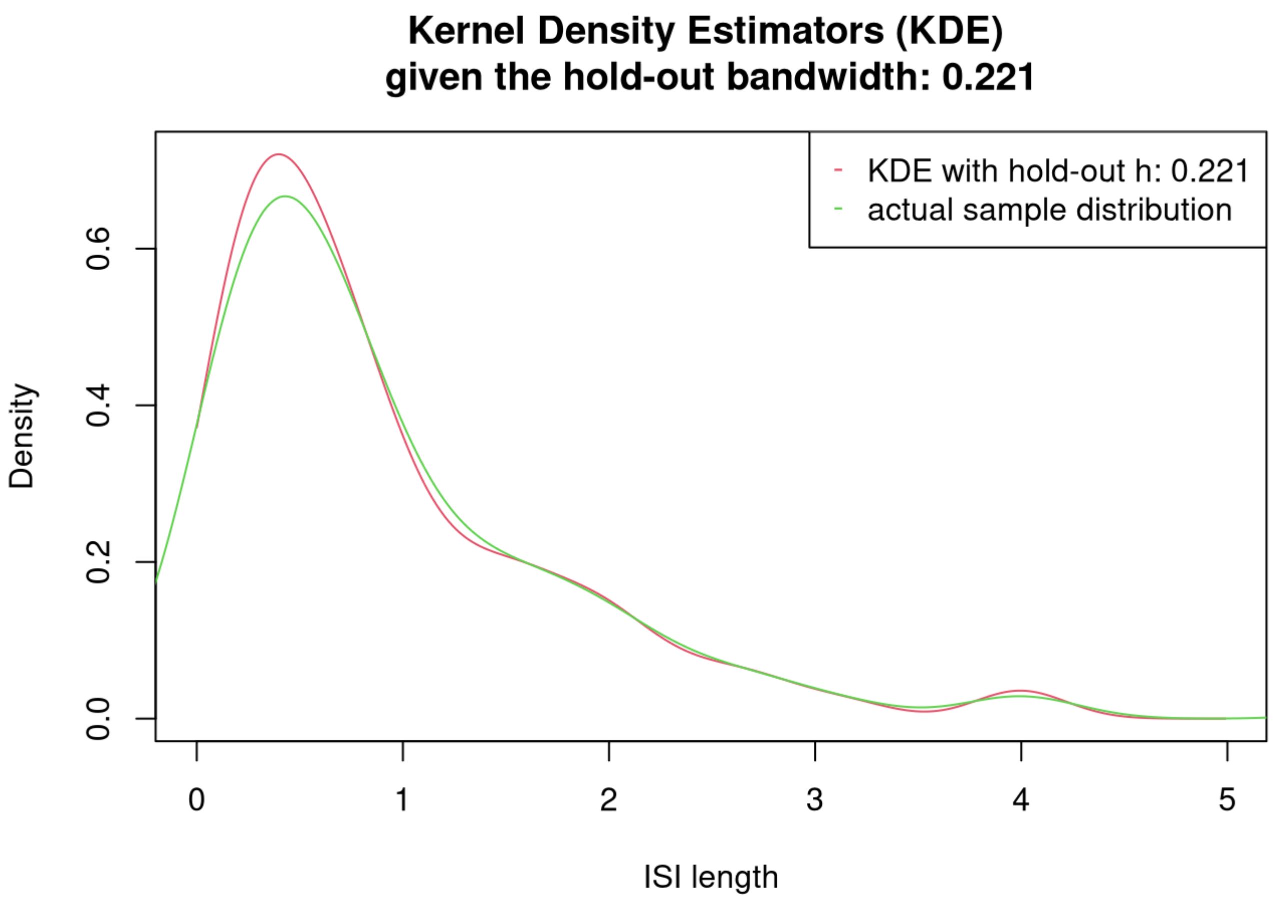
After hold-out validation, the lowest contrast was achieved on the simulation of the exponential distribution $\mathcal{E}(1)$ with the bandwidth parameter h set to:

```
best_h = h[which(contrasts==min(contrasts))]
best_h
```

```
## [1] 0.221
```

With the current best parameter h given the hold-out validation, let's compute and plot the kernel estimator again based on it:

```
# Computes the kernel estimates again with the selected h
estimator = kernel_estimator(ISI, u, best_h)

# Plots the distribution
plot_kde_vs_sample_dist(estimator, ISI, best_h, val_method="hold-out")
```



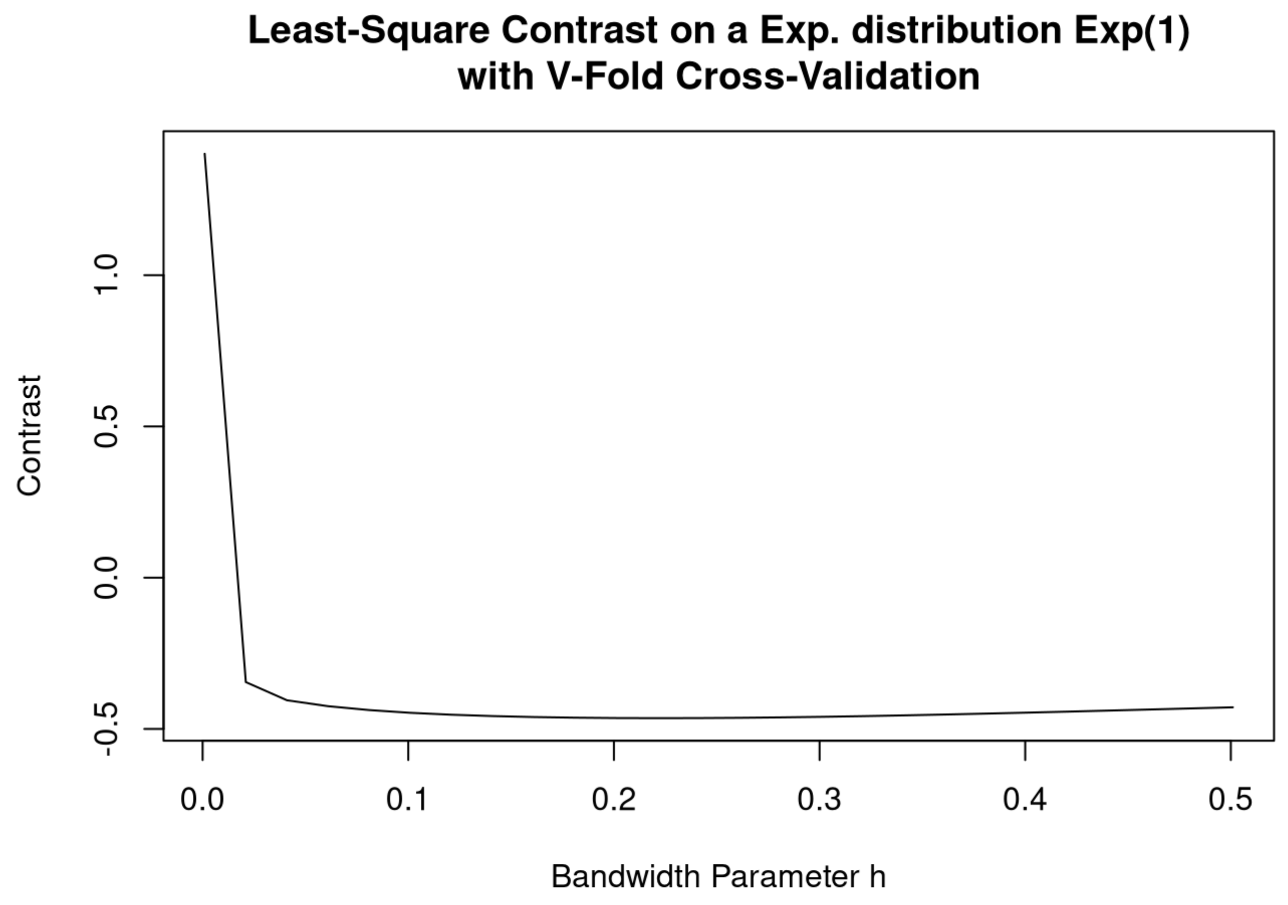
Step E

In order to implement our cross-validation, let's implement a specifically-designed loop function to iterate over both the available data and the set of possible bandwidth parameters h.

```
v_fold_cross_validation <- function(ISI, h, v, u, u_stepsize) {
  # Implements a v-fold cross-validation process over a ISI dataset
  # in order to find the best bandwidth parameter h out of a candidate list
  #
  # Creates v-folds out of the input ISI dataset
  n = length(ISI)
  folds = split(ISI,ceiling(seq_along(ISI)/round(length(ISI)/v)))
  contrasts = c()
  for (test_fold in 1:v) {
    # Computes the data to be used for training and for testing
    train_folds = c(1:v)[c(1:v)!=test_fold]
    train_ISI = as.vector(unlist(folds[train_folds]))
    test_ISI = as.vector(unlist(folds[test_fold]))
    # Computes the contrasts for each fold pass
    contrast = apply(h,1,function(x) {
      contrast_least_squares(train_ISI, u, u_stepsize, x, test_ISI)
    })
    # Records the computed contrats
    contrasts = cbind(contrasts, contrast)
  }
  1/v*apply(contrasts,1,sum)
}
```

With a number of folds to 10, let’s compute and visualize the V-fold Cross-Validation:

```
# Declares v
v = 10

# Computes the v_fold_contrasts
v_fold_contrasts = v_fold_cross_validation(ISI, h, 10, u, u_stepsize)

# Plots the given contrasts for the set of bandwidths h
plot(h, v_fold_contrasts, type="l",
     xlab="Bandwidth Parameter h", ylab="Contrast",
     main="Least-Square Contrast on a Exp. distribution Exp(1)\nwith V-Fold Cross-Validation")
```



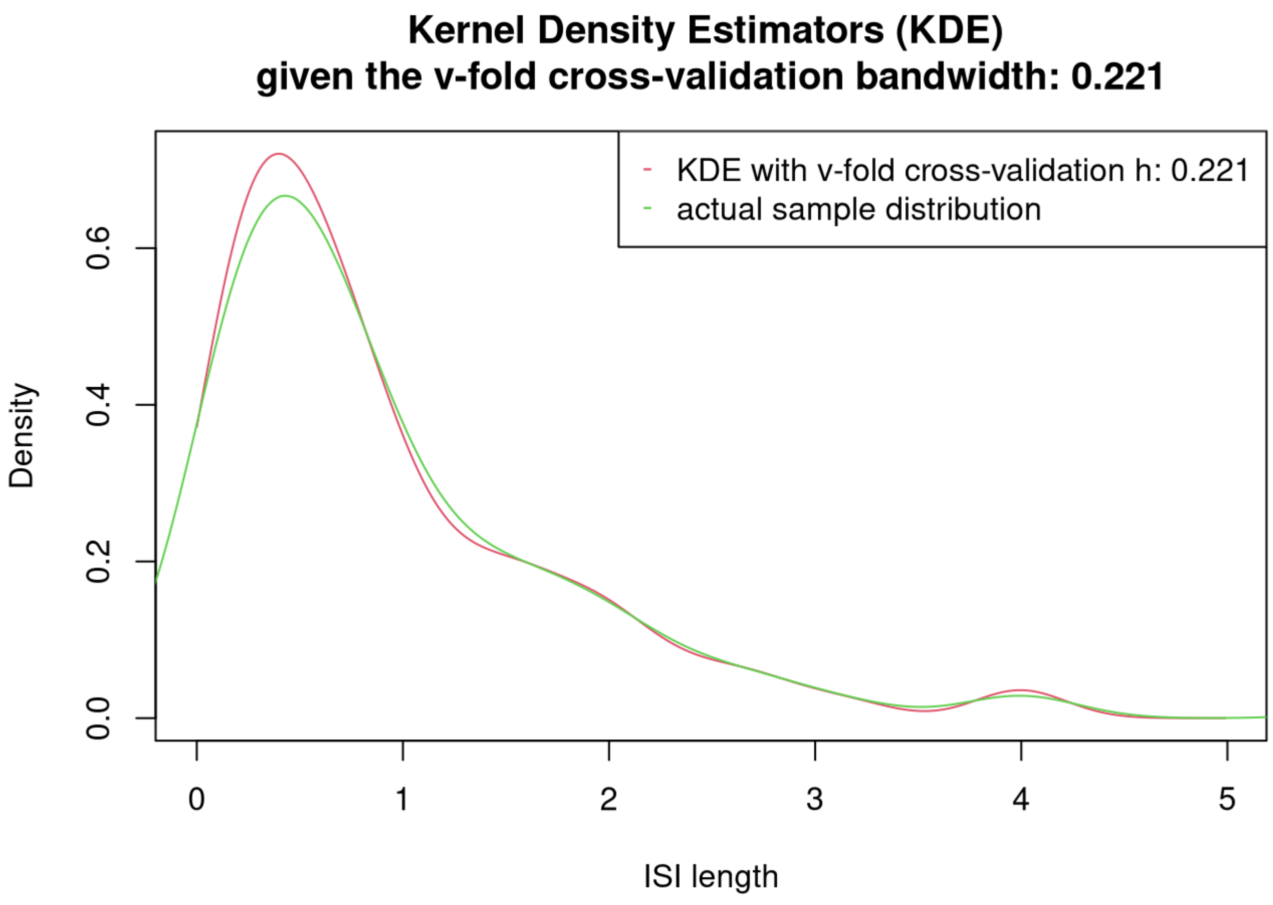
After v-fold cross-validation, the lowest contrast was achieved on the simulation of the exponential distribution $\mathcal{E}(1)$ with the bandwidth parameter h set to:

```
best_h = h[which(v_fold_contrasts==min(v_fold_contrasts))]
best_h
```

```
## [1] 0.221
```

With the current best parameter h given the hold-out validation, let's compute and plot the kernel estimator again based on it:

```
# Computes the kernel estimates again with the selected h
estimator = kernel_estimator(ISI, u, best_h)

# Plots the distribution
plot_kde_vs_sample_dist(estimator, ISI, best_h, val_method="v-fold cross-validation")
```

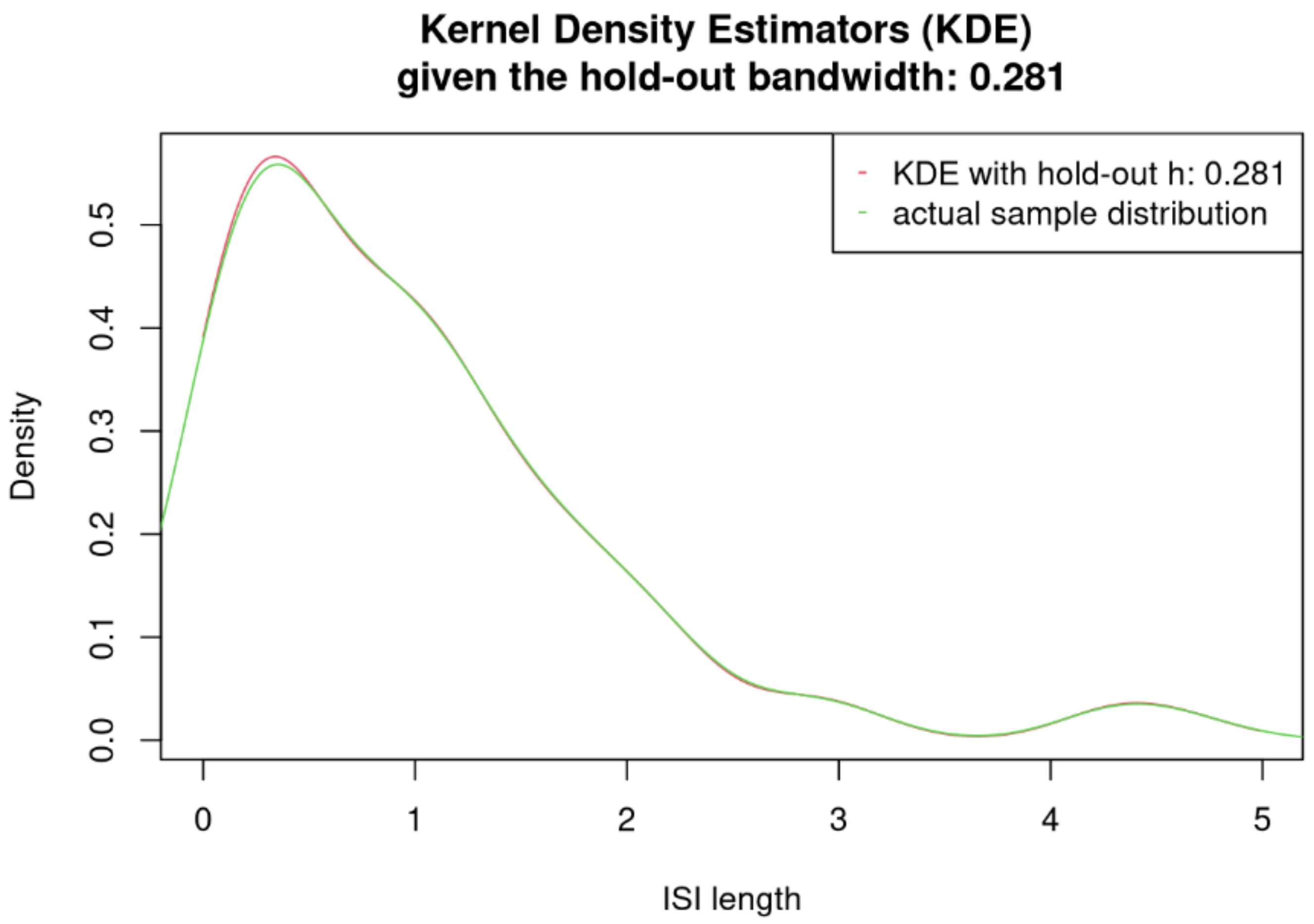


Note on Step D and E

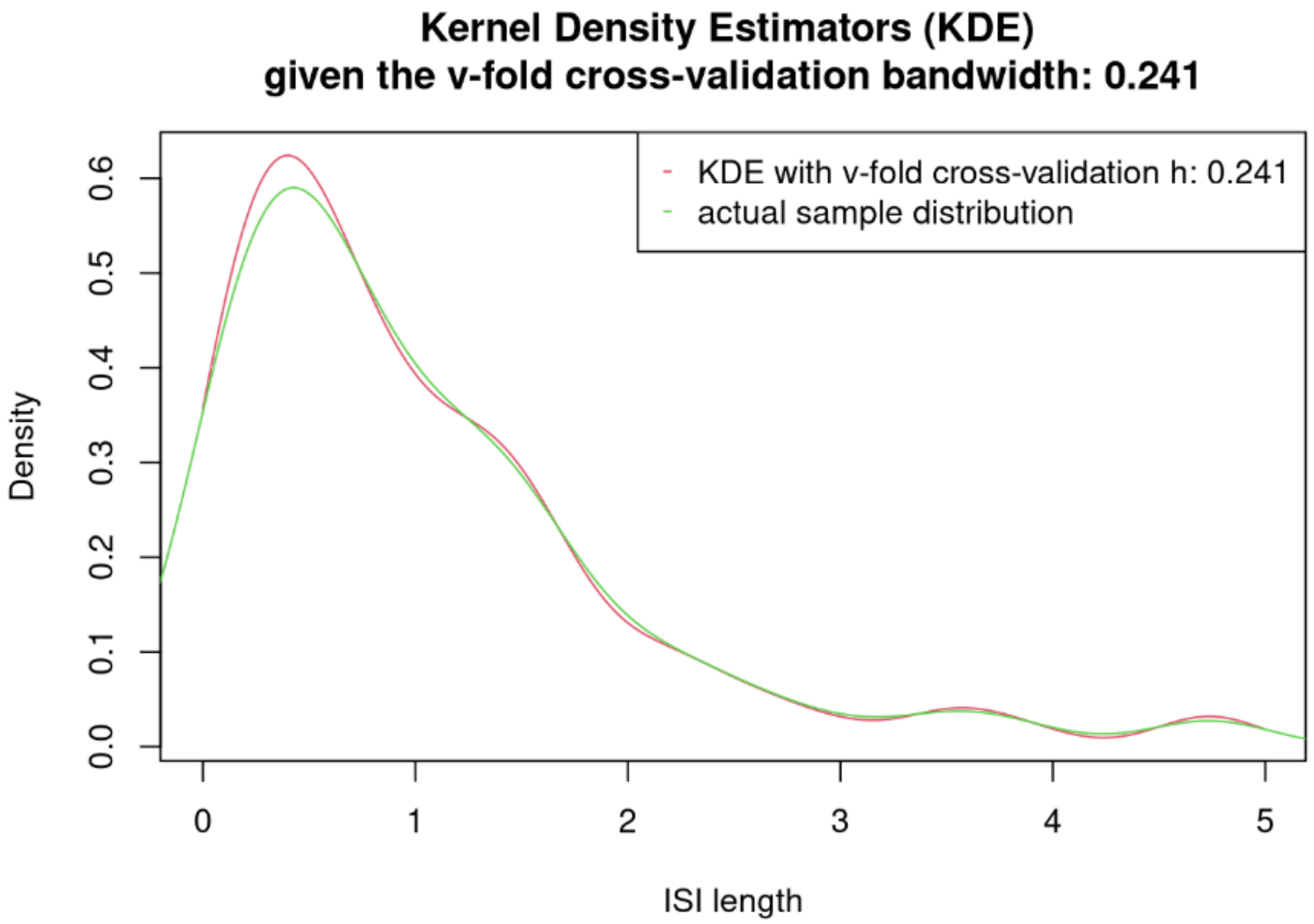
It is observed that, by visual inspection of the plots, the estimations with the hold-out and v-fold cross-validation perform better than the first method (which did not use validation).

In the current implementation, it seems that the hold-out method tends to provide an equivalent or better estimation than v-fold cross-validation (i.e. with this specific selection of h bandwidth and u range, hold-out seems to offer a better h in general).

However, by running the above process several times, it happens that the hold-out and v-fold cross-validation methods can find very fitting approximations (*though not always during the same simulation*), which are very close to the actual sample density:



best_case



best_case2

2 - Parametric Bootstrap

Overview

Let’s implement parametric bootstrap.

A – Let’s simulate $n = 50$ exponential variables with parameter $\theta_0 = 2$. This will be considered as observations.

B – Let’s compute the Maximum Likelihood Estimator (MLE) $\hat{\theta}_{obs}$ of θ for $n = 50$ IID ISI modeled by an exponential distribution.

C – By simulating $N_simu = 10000$ times n exponential variables with parameter $\theta_0 = 2$, let’s compute an approximate classical confidence interval IC_1 on 4$ with confidence 95%.

Of note: To do that, one can for instance compute $D_i = |\hat{\theta}_i - \theta_0|$ on each simulation and decide that, with a c. 95% probability, the distance between $\hat{\theta}$ and θ_0 is less than the 95% empirical quantile of D_i. this cannot be done on real data since θ_0 is not known.

D – By bootstrapping the data the parametric approach, let’s compute a bootstrap confidence interval IC2. To do so, n exponential variables are simulated N times with parameter $\theta = \hat{\theta}$ and do as before with respect to the simulation.

E – Let’s compare IC_1 and IC_2 for various size of N. Let’s also explain why IC_2 cannot converge to IC_1 when $N \to +\infty$, even if it is a pretty good approximation.

Step A

```
# Declares parameters
n = 50
theta_0 = 2

# Simulates
observations = rexp(n, theta_0)
```

Step B

The parametric approach informs that the MLE for IID ISI modeled by an exponential distribution is the following: $\hat{\theta}_{obs} = \frac{1}{\bar{X}}$ with $\bar{X}$ the sample mean of the observations. The proof can be found here: StatLect (https://www.statlect.com/fundamentals-of-statistics/exponential-distribution-maximum-likelihood) (last accessed, Dec 5, 2021).

```
hat_theta_obs <- function(observations){
  # Computes the MLE of observations following an exponential distribution
  1/(mean(observations))
}

# Computes and prints the MLE of the generated 50 observations
hat_theta = hat_theta_obs(observations)
hat_theta
```

```
## [1] 2.621475
```

Step C

Let’s compute the 95% confidence interval IC_1 of θ. To do that, IC_1 is derived by preliminarilly computing the 95% quantile of D:

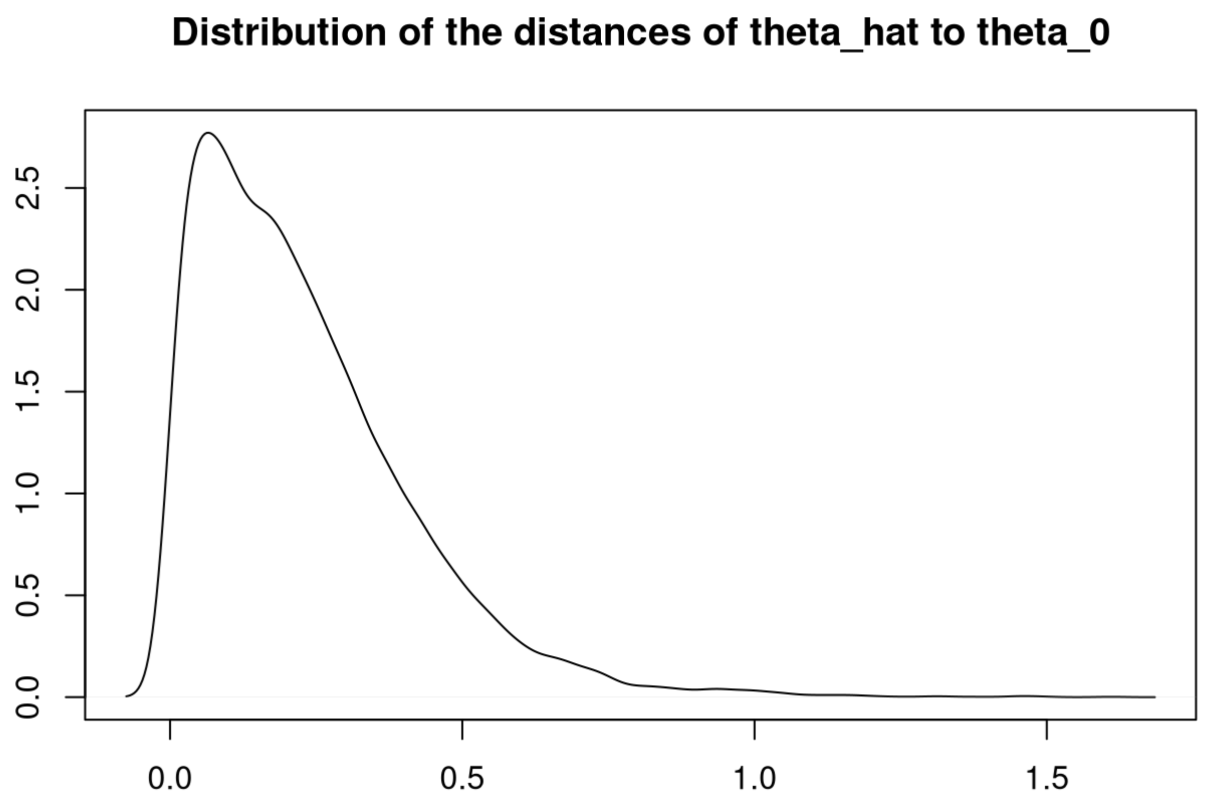
$$\forall i \in \{1, \dots, n_{simulations}\},\ D_i = |\hat{\theta}_i - \theta_0|$$

.

```
compute_distances <- function(N, n, theta_0) {
  # Computes the MLE of each simuation
  hat_theta_sims = apply(matrix(1:n_simulations), 1,
                         function(x) {hat_theta_obs(rexp(n, theta_0))})
  # Computes the difference/distance between each MLE and the
  # real parameter theta_0
  abs(theta_0-hat_theta_sims)
}

# Declares variables
n_simulations = 10000

distances = compute_distances(n_simulations, n, theta_0)
```

With $D_i, \forall i \in \{1, \ldots, n_{simulations}\}$, the distribution of D can be plotted for visualization purposes:

```
plot(density(distances), xlab="", ylab="",
     main="Distribution of the distances of theta_hat to theta_0")
```



Let's compute the 95% quantile of D and can rework the 95% confidence interval on θ.

```
compute_95CI <- function(distances, theta_0, print_msg=F) {
  # Computes the 95% quantile of D
  quantile95_distances = quantile(distances, probs=c(0.95))
  # Computes and prints the resulting CI1 of theta
  CI = c(theta_0-quantile95_distances, theta_0+quantile95_distances)
  CI = t(as.matrix(setNames(CI, c("2.5%", "97.5%"))))
  if (print_msg) {
    cat("The classical 95% confidence interval for theta is:\n", CI)
  }
  CI
}

IC = compute_95CI(distances, theta_0, T)
```

```
## The classical 95% confidence interval for theta is:
##  1.425545 2.574455
```

Step D

Let's reuse the content used in steps A and B for the following steps. To perform the parametric bootstrapping, the number of simulation is arbitrarily set to $N = 10000$, as with Step C.

```
# Declares or recalls the parameters
N = 10000
n= 50

# Declares the bootstrap version of compute_distances
compute_distances_bootstrap <- function(N, n, observations) {
  # Computes the MLE of each simuation
  hat_theta_sims = apply(matrix(1:n_simulations), 1,
                         function(x) {hat_theta_obs(
                           sample(observations, n, replace=T)
                         )})
  # Computes the difference/distance between each MLE and the
  # real parameter theta_0
  abs(theta_0-hat_theta_sims)
}

# Substitutes our previously computed hat_theta for theta_0 in the
# previously declared function setup
distances = compute_distances_bootstrap(n_simulations, n, observations)
IC2 = compute_95CI(distances, hat_theta, T)
```

```
## The classical 95% confidence interval for theta is:
##  1.343358 3.899593
```

Step E

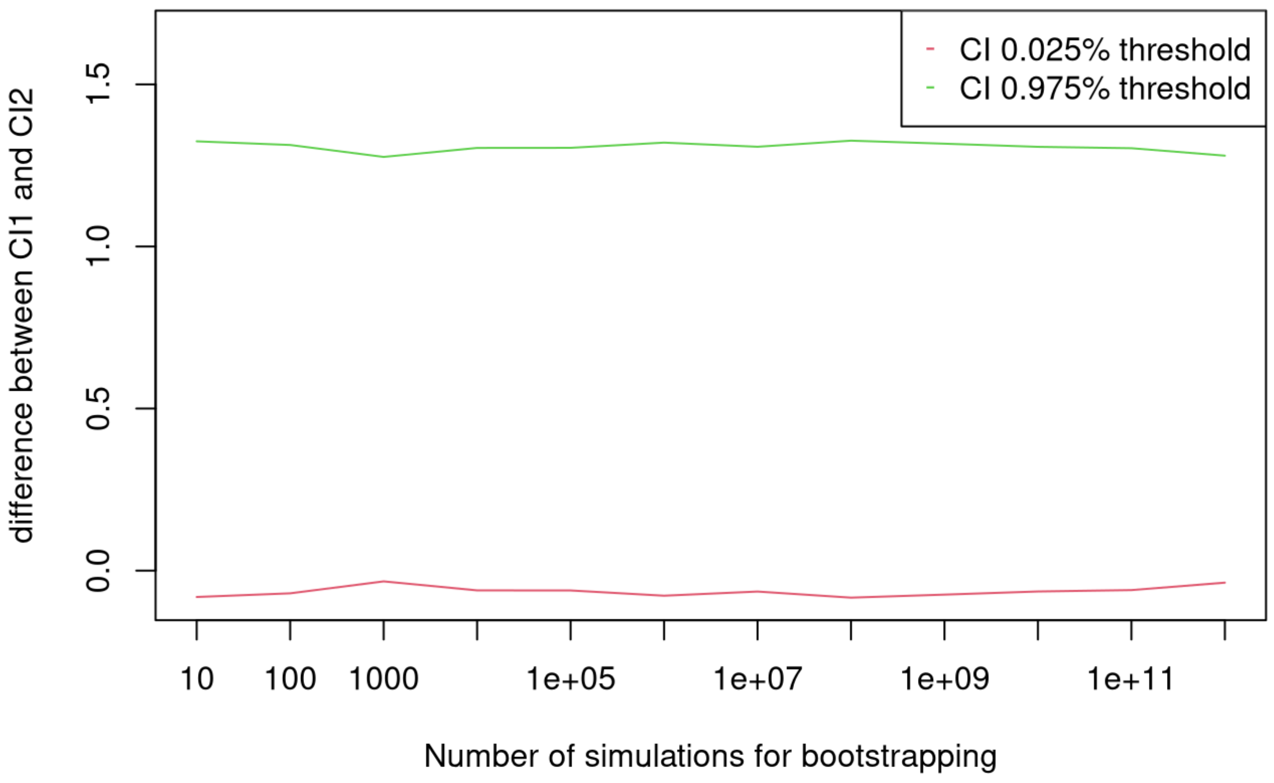
Let's reuse the previous process, varying N from 10 to $1e12$ by orders of magnitude.

```
# Declares the range for N_simulations
N_range = matrix(apply(matrix(1:12), 1, function(x){10^x}))

# Declares the composition function to compute CIs
compute_ci <- function(n_simulations, n, theta, observations, bootstrap=F) {
  if (bootstrap) {
    compute_95CI(compute_distances_bootstrap(n_simulations, n, observations), theta)
  } else {
    compute_95CI(compute_distances(n_simulations, n, theta), theta)
  }
}

# Computes the CI1 and CI2 for the given N range
cis = apply(N_range, 1, function(x){c(compute_ci(x, n, theta_0),
                                      compute_ci(x, n, hat_theta, observations, T))})
cis = t(cis)
row.names(cis) = N_range
colnames(cis) = c("CI1_0.025","CI1_0.975","CI2_0.025","CI2_0.975")
cis
```

```
##        CI1_0.025 CI1_0.975 CI2_0.025 CI2_0.975
## 10      1.420997  2.579003  1.339633  3.903318
## 100     1.418556  2.581444  1.348483  3.894468
## 1000    1.409899  2.590101  1.376647  3.866304
## 10000   1.420501  2.579499  1.359507  3.883444
## 1e+05   1.419712  2.580288  1.358442  3.884509
## 1e+06   1.426833  2.573167  1.349449  3.893502
## 1e+07   1.414519  2.585481  1.349915  3.893036
## 1e+08   1.424935  2.575065  1.341550  3.901401
## 1e+09   1.415983  2.584017  1.341926  3.901025
## 1e+10   1.420143  2.579857  1.355757  3.887194
## 1e+11   1.406336  2.593664  1.346234  3.896717
## 1e+12   1.412213  2.587787  1.375075  3.867876
```

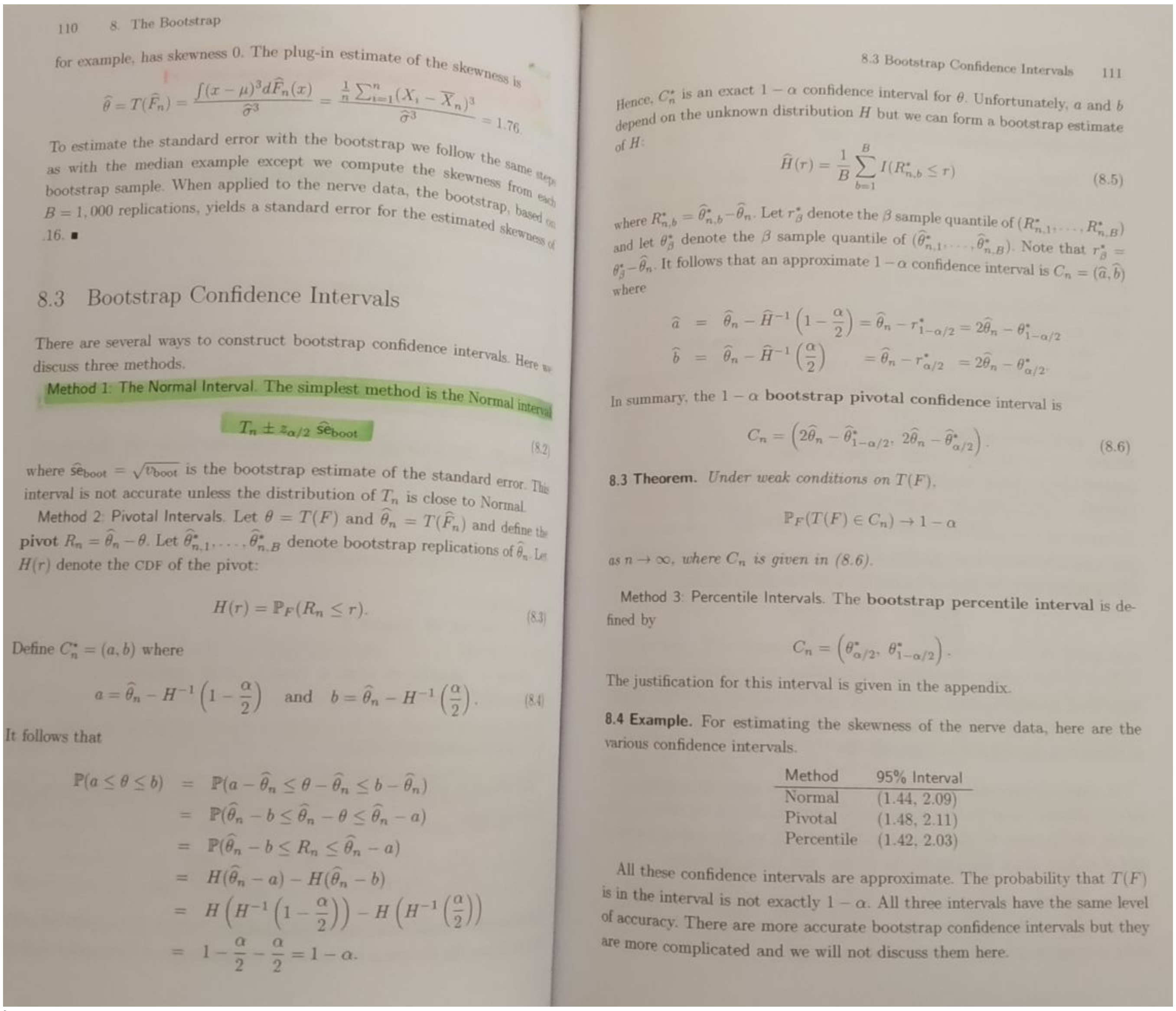
Let's compute the difference between CI_1 and CI_2's confidence interval lower and upper bounds. Those differences are then plotted:

```
differences = c(cis[,3]-cis[,1], cis[,4]-cis[,2])
differences = matrix(differences, ncol=2)

plot(differences[,1],
     ylim = c(min(differences), max(differences)+abs(0.25*max(differences))),
     type="l", col=2, xaxt="n", ylab="difference between CI1 and CI2",
     xlab="Number of simulations for bootstrapping")
axis(1, at=c(1:12), labels=as.vector(N_range))
lines(differences[,2], type="l", col=3)
legend("topright", legend=c("CI 0.025% threshold", "CI 0.975% threshold"),
       col=c(2,3), pch="-")
```



It is evidenced that there is no apparent convergence of the two confidence intervals.

This is due to the theory of the bootstrap pivotal confidence interval (See picture from *All of Statistics*, by L. Wasserman, p111) that states that the boostrap confidence interval will converge towards that given interval C_n, but that confidence interval is not the original (but is close). Working with bootstrapped values, the underlying distribution is not $\exp(\theta_0)$ but the bootstrapped one, which is never exactly the same.


110 8. The Bootstrap

for example, has skewness 0. The plug-in estimate of the skewness is

$$\widehat{\theta} = T(\widehat{F}_n) = \frac{\int (x-\mu)^3 d\widehat{F}_n(x)}{\widehat{\sigma}^3} = \frac{\frac{1}{n}\sum_{i=1}^n (X_i - \overline{X}_n)^3}{\widehat{\sigma}^3} = 1.76.$$

To estimate the standard error with the bootstrap we follow the same steps as with the median example except we compute the skewness from each bootstrap sample. When applied to the nerve data, the bootstrap, based on $B = 1,000$ replications, yields a standard error for the estimated skewness of .16. ■

8.3 Bootstrap Confidence Intervals

There are several ways to construct bootstrap confidence intervals. Here we discuss three methods.

Method 1: The Normal Interval. The simplest method is the Normal interval

$$T_n \pm z_{\alpha/2}\,\widehat{\mathsf{se}}_{\text{boot}} \tag{8.2}$$

where $\widehat{\mathsf{se}}_{\text{boot}} = \sqrt{v_{\text{boot}}}$ is the bootstrap estimate of the standard error. This interval is not accurate unless the distribution of T_n is close to Normal.

Method 2: Pivotal Intervals. Let $\theta = T(F)$ and $\widehat{\theta}_n = T(\widehat{F}_n)$ and define the **pivot** $R_n = \widehat{\theta}_n - \theta$. Let $\widehat{\theta}^*_{n,1}, \ldots, \widehat{\theta}^*_{n,B}$ denote bootstrap replications of $\widehat{\theta}_n$. Let $H(r)$ denote the CDF of the pivot:

$$H(r) = \mathbb{P}_F(R_n \le r). \tag{8.3}$$

Define $C_n^* = (a, b)$ where

$$a = \widehat{\theta}_n - H^{-1}\left(1 - \frac{\alpha}{2}\right) \quad \text{and} \quad b = \widehat{\theta}_n - H^{-1}\left(\frac{\alpha}{2}\right). \tag{8.4}$$

It follows that

$$\begin{aligned} \mathbb{P}(a \le \theta \le b) &= \mathbb{P}(a - \widehat{\theta}_n \le \theta - \widehat{\theta}_n \le b - \widehat{\theta}_n) \\ &= \mathbb{P}(\widehat{\theta}_n - b \le \widehat{\theta}_n - \theta \le \widehat{\theta}_n - a) \\ &= \mathbb{P}(\widehat{\theta}_n - b \le R_n \le \widehat{\theta}_n - a) \\ &= H(\widehat{\theta}_n - a) - H(\widehat{\theta}_n - b) \\ &= H\left(H^{-1}\left(1 - \frac{\alpha}{2}\right)\right) - H\left(H^{-1}\left(\frac{\alpha}{2}\right)\right) \\ &= 1 - \frac{\alpha}{2} - \frac{\alpha}{2} = 1 - \alpha. \end{aligned}$$

8.3 Bootstrap Confidence Intervals 111

Hence, C_n^* is an exact $1-\alpha$ confidence interval for θ. Unfortunately, a and b depend on the unknown distribution H but we can form a bootstrap estimate of H:

$$\widehat{H}(r) = \frac{1}{B}\sum_{b=1}^{B} I(R^*_{n,b} \le r) \tag{8.5}$$

where $R^*_{n,b} = \widehat{\theta}^*_{n,b} - \widehat{\theta}_n$. Let r^*_β denote the β sample quantile of $(R^*_{n,1}, \ldots, R^*_{n,B})$ and let θ^*_β denote the β sample quantile of $(\widehat{\theta}^*_{n,1}, \ldots, \widehat{\theta}^*_{n,B})$. Note that $r^*_\beta = \theta^*_\beta - \widehat{\theta}_n$. It follows that an approximate $1-\alpha$ confidence interval is $C_n = (\widehat{a}, \widehat{b})$ where

$$\begin{aligned} \widehat{a} &= \widehat{\theta}_n - \widehat{H}^{-1}\left(1 - \frac{\alpha}{2}\right) = \widehat{\theta}_n - r^*_{1-\alpha/2} = 2\widehat{\theta}_n - \theta^*_{1-\alpha/2} \\ \widehat{b} &= \widehat{\theta}_n - \widehat{H}^{-1}\left(\frac{\alpha}{2}\right) \quad = \widehat{\theta}_n - r^*_{\alpha/2} \quad = 2\widehat{\theta}_n - \theta^*_{\alpha/2}. \end{aligned}$$

In summary, the $1-\alpha$ **bootstrap pivotal confidence** interval is

$$C_n = \left(2\widehat{\theta}_n - \widehat{\theta}^*_{1-\alpha/2},\; 2\widehat{\theta}_n - \widehat{\theta}^*_{\alpha/2}\right). \tag{8.6}$$

8.3 Theorem. *Under weak conditions on $T(F)$,*

$$\mathbb{P}_F(T(F) \in C_n) \to 1 - \alpha$$

as $n \to \infty$, where C_n is given in (8.6).

Method 3: Percentile Intervals. The **bootstrap percentile interval** is defined by

$$C_n = \left(\theta^*_{\alpha/2},\; \theta^*_{1-\alpha/2}\right).$$

The justification for this interval is given in the appendix.

8.4 Example. For estimating the skewness of the nerve data, here are the various confidence intervals.

Method	95% Interval
Normal	(1.44, 2.09)
Pivotal	(1.48, 2.11)
Percentile	(1.42, 2.03)

All these confidence intervals are approximate. The probability that $T(F)$ is in the interval is not exactly $1-\alpha$. All three intervals have the same level of accuracy. There are more accurate bootstrap confidence intervals but they are more complicated and we will not discuss them here.

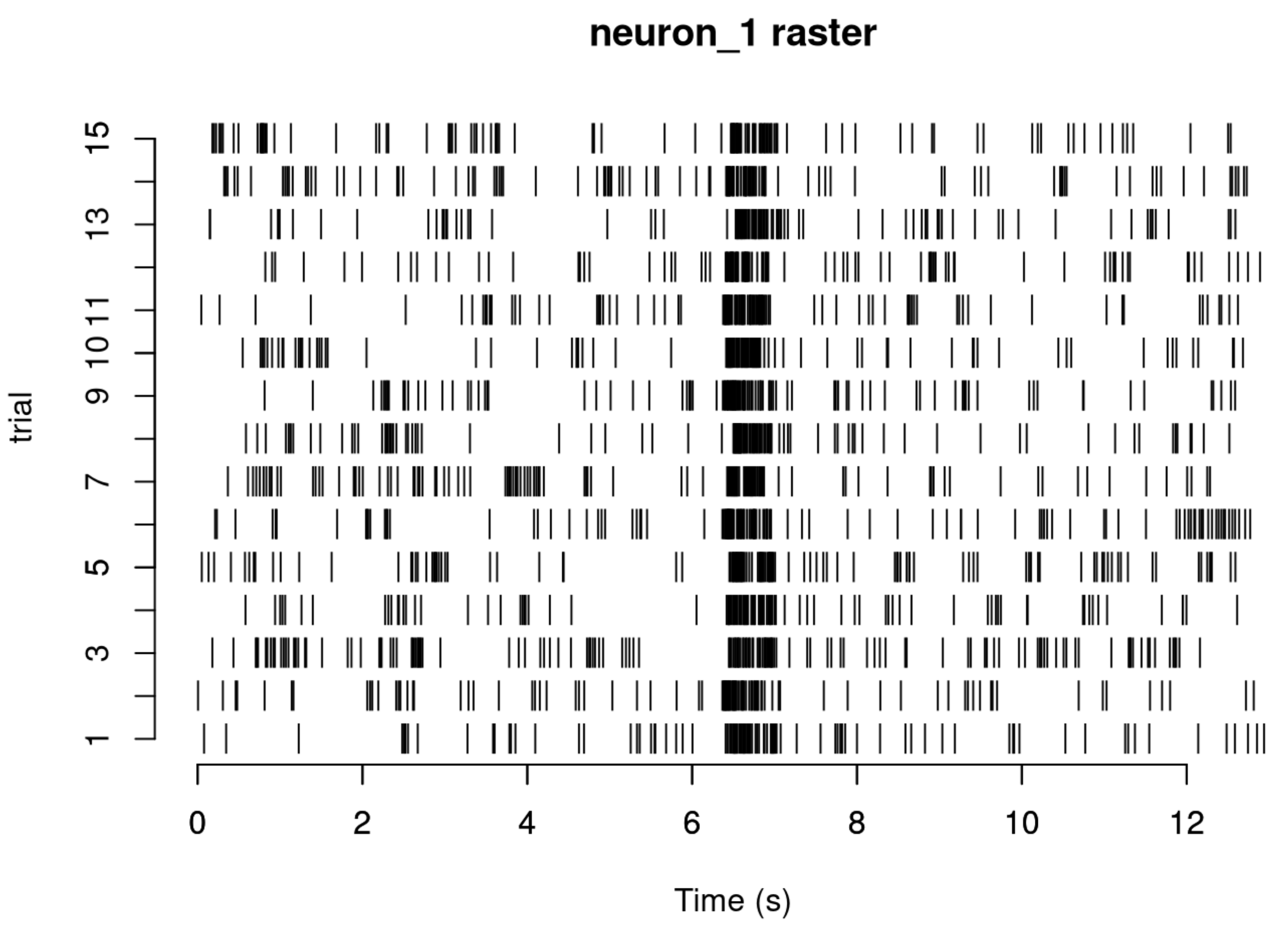
bootstrap

Step F

```
#install.packages("STAR")
library(STAR)
```

Let’s import the data:

```
data(e070528citronellal)
neuron_1 = e070528citronellal[["neuron 1"]]

plot(neuron_1)
```



Given prior knowledge, the periods characterizing the behavior of the neuron are defined as such:

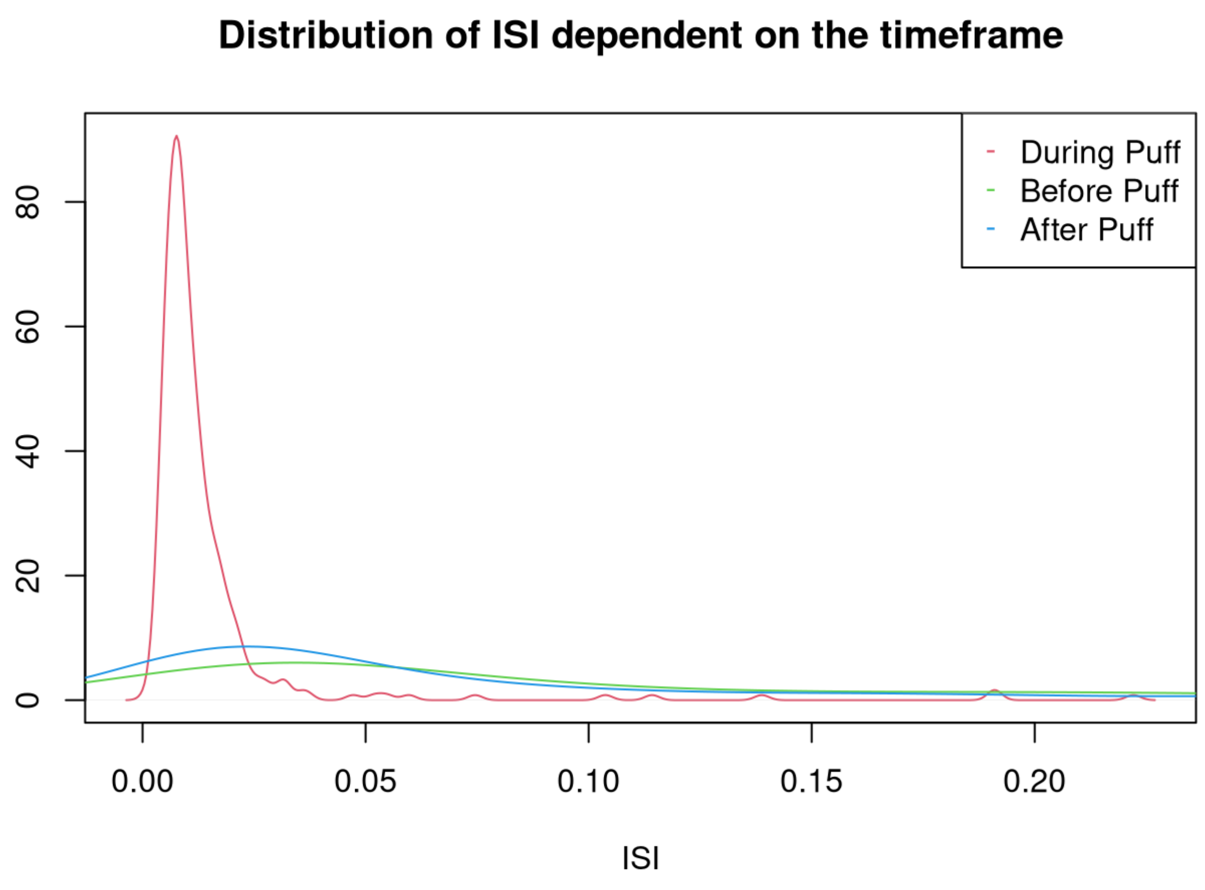
- Before the puff: $t \in [0, 6.14]$
- During the puff: $t \in]6.14, 6.64[$
- After the puff: $t \in [6.64, c.13]$

Let's retrieve the 15 simulations' spike data into three sets as vectors.

```
spike_data_extract <- function(neuron_data, lower=NULL, upper=NULL){
  # Extracts spike data in a given time frame and return
  # the corresponding ISI
  ISI=c()
  for (sim in 1:length(neuron_data)){
    sim_spikes = as.vector(neuron_1[[sim]])
    if (is.null(lower)){
      indexes = which((sim_spikes>=upper))
    } else if (is.null(upper)) {
      indexes = which((sim_spikes<=lower))
    } else {
      indexes = which((sim_spikes>lower)&(sim_spikes<upper))
    }
    ISI = c(ISI, diff(sim_spikes[indexes]))
  }
  return(list("ISI"=ISI,
              "theta_hat"=1/(mean(ISI)-min(ISI))))
}

# Retrieves the data per period
data_before = spike_data_extract(neuron_1, 6.14, NULL)
data_during = spike_data_extract(neuron_1, 6.14, 6.65)
data_after  = spike_data_extract(neuron_1, NULL, 6.65)
```

Let's plot the extracted datasets against each other:

```r
plot(density(data_during$ISI), type="l", col=2, xlab="ISI", ylab="",
     "Distribution of ISI dependent on the timeframe")
lines(density(data_before$ISI), type="l", col=3)
lines(density(data_after$ISI), type="l", col=4)
legend("topright", legend=c("During Puff", "Before Puff", "After Puff"),
        col=c(2,3,4),pch="-")
```



It is visually verifiable that there is a discrepancy between the distributions. To show that statistically, let's first start by declaring the following functions:

```r
compute_distances_bootstrap_bonferroni <- function(N, observations, theta) {
  # Computes the MLE of each simuation
  hat_theta_sims = apply(matrix(1:n_simulations), 1,
                       function(x) {hat_theta_obs(
                         sample(observations, n, replace=T)
                       )})
  # Computes the difference/distance between each MLE and the
  # real parameter theta_0
  abs(theta-hat_theta_sims)
}

compute_95CI_bonferroni <- function(distances, theta, k_tests, print_msg=F) {
  # Computes the 95% quantile of the distances between a boostrapped sample
  # of ISI, given a Bonferroni correction
  quantile95_distances = quantile(distances, probs=c(1-0.5/k_tests))
  # Computes and prints the resulting CI1 of theta
  CI = c(theta-quantile95_distances, theta+quantile95_distances)
  CI = t(as.matrix(setNames(CI, c("2.5%", "97.5%"))))
  if (print_msg) {
    cat("The classical 95% confidence interval",
        "with Bonferroni correction for theta is:\n", CI,"\n")
  }
  CI
}
```

Let's now compute the 95% confidence interval with Bonferroni correction. Here the number of tests of level $\alpha = 0.05$ is 3 since one confidence interval per dataset (before, during, after) is computed. For the bootstrapping, let's reuse the previous number of simulations $N = 10000$.

```
N=10000

distances = compute_distances_bootstrap_bonferroni(N, data_before$ISI, data_before$theta_hat)
ICbefore = compute_95CI_bonferroni(distances, data_before$theta_hat, 3, T)
```

```
## The classical 95% confidence interval with Bonferroni correction for theta is:
##  4.509311 8.118943
```

```
distances = compute_distances_bootstrap_bonferroni(N, data_during$ISI, data_during$theta_hat)
ICduring = compute_95CI_bonferroni(distances, data_during$theta_hat, 3, T)
```

```
## The classical 95% confidence interval with Bonferroni correction for theta is:
##  53.44468 96.019
```

```
distances = compute_distances_bootstrap_bonferroni(N, data_after$ISI, data_after$theta_hat)
ICafter = compute_95CI_bonferroni(distances, data_after$theta_hat, 3, T)
```

```
## The classical 95% confidence interval with Bonferroni correction for theta is:
##  5.989016 10.58613
```

As such, it is evidenced that there is a clear distinction between the "before" and "after" periods and the "during" period as they do not overlap. Similarly, it is uncertain whether or not the "before" and "after" periods are distinct in terms of behavior as their confidence interval do overlap.

3 - Non-Parametric Boostrap

Overview

Reusing the previous data of the STAR package, let's perform non-parametric bootstrap (with firing rates instead of ISI). Let's compute to compute non parametric bootstrap confidence intervals on the firing rates. A firing rate is defined by the average number of spikes per second.

A – Let's compute an estimation of the firing rate for each of the three periods for the experiment `citronellal` and for each of the trials.

B – Let's code a function for computing the bootstrap confidence interval on the mean at a given level α. Then let's test it on simulated data to see if it works.

C – Let's apply it on the estimated firing rates of each period in **A** and see if it can be concluded non-parametrically on a difference between the three periods in terms of firing rates.

Step A

To compute an estimation of the firing rate, let's take the average firing rate per second over each of the three periods outlined in the previous section.

Let's first declare the corresponding function:

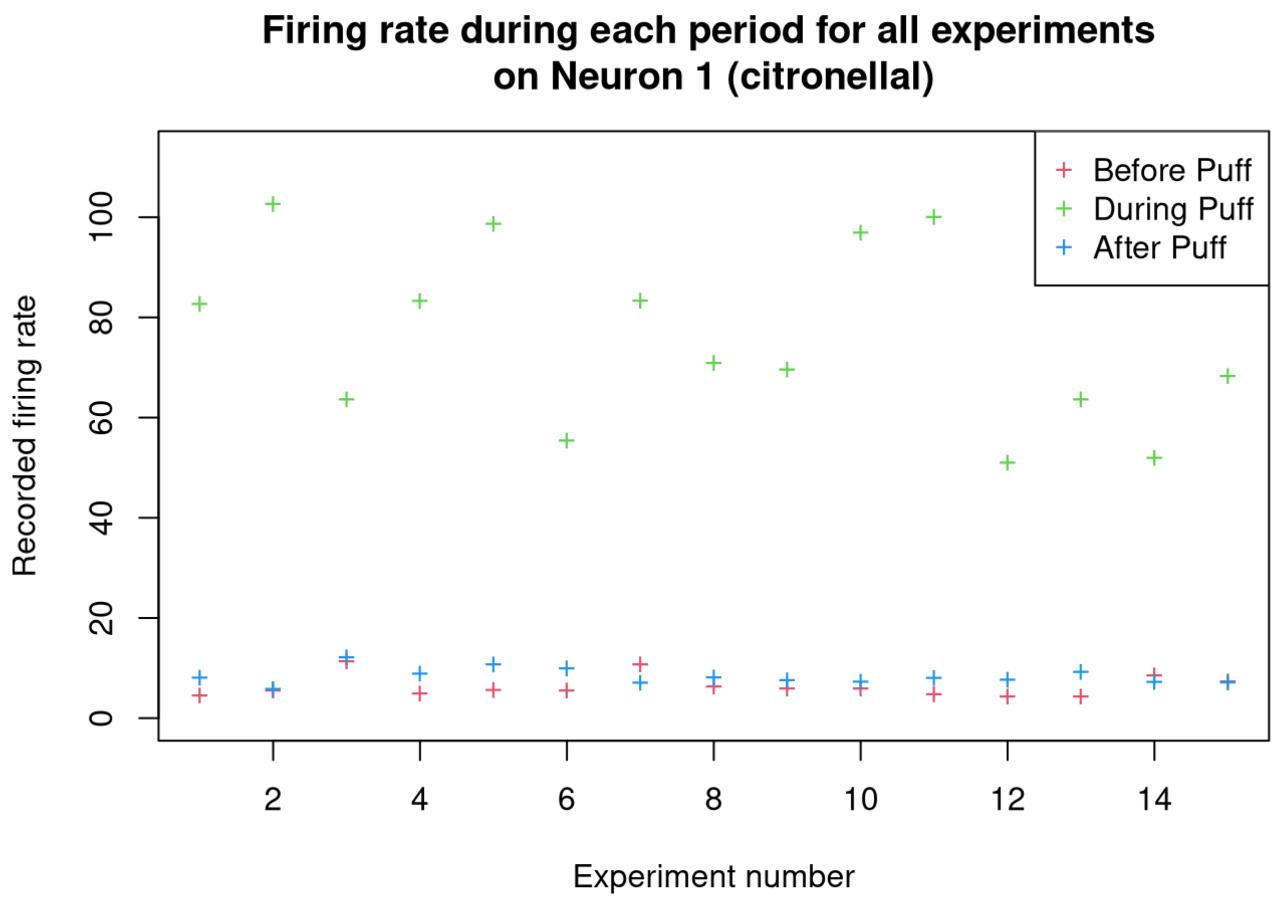
```
firing_rate_extract <- function(neuron_data, lower=NULL, upper=NULL){
  # Extracts spike data in a given time frame and return
  # the corresponding ISI
  #
  # Data extraction function
  spike_extract <- function(sim){
    sim_spikes = as.vector(neuron_1[[sim]])
    if (is.null(lower)){
      indexes = which((sim_spikes>=upper))
    } else if (is.null(upper)) {
      indexes = which((sim_spikes<=lower))
    } else {
      indexes = which((sim_spikes>lower)&(sim_spikes<upper))
    }
    sim_spikes[indexes]
  }
  # Extracts the spike data
  spikes=c()
  n_simulations = as.matrix(c(1:length(neuron_data)))
  spikes = apply(n_simulations, 1, spike_extract)
  # Computes the firing rate for each of the 3 periods and 15 simulations
  firing_rates = apply(n_simulations, 1,
                        function(x){
                          length(spikes[[x]])/(max(spikes[[x]])-min(spikes[[x]]))
                       })
  return(list("spikes"=spikes,
              "firing_rates"=firing_rates))
}
```

Let's extract the data and compute the corresponding firing rates.

```
data_before = firing_rate_extract(neuron_1, 6.14, NULL)
data_during = firing_rate_extract(neuron_1, 6.14, 6.65)
data_after  = firing_rate_extract(neuron_1, NULL, 6.65)
```

With the firing rate computed, let's visualize them for each simulation.

```
plot(data_before$firing_rates, pch="+",
     ylim=c(0,max(data_during$firing_rates)+10), col=2,
     xlab="Experiment number", ylab="Recorded firing rate",
     main=paste("Firing rate during each period for all experiments",
                "\non Neuron 1 (citronellal)")
)
points(data_during$firing_rates, pch="+", col=3)
points(data_after$firing_rates, pch="+", col=4)
legend("topright", legend=c("Before Puff", "During Puff", "After Puff"),
        col=c(2,3,4),pch="+")
```



Step B

Let’s build a non-parametric bootstrap function for the mean confidence interval:

```r
mean_CI_bootstrap <- function(data, n_simulations, alpha, print_msg=F) {
  # Performs a bootstrap of the mean of a given data
  #
  # Computes preliminary parameters
  n = length(data)
  sample_mean = mean(data)
  # Bootstraps the data with unknown mean
  bootstrapped_data = t(apply(
    matrix(c(1:n_simulations)), 1,
    function(x) {sample(data, n, replace=T)}
  ))
  # Computes a sample mean for each bootstrapped sample
  bootstrapped_means = apply(bootstrapped_data, 1, mean)
  # Computes the distance between bootstrapped means and sample mean
  bootstrapped_dists = bootstrapped_means - sample_mean
  # Computes the bootstrapped confidence interval at level 1-alpha
  q_alpha_div_2 = quantile(bootstrapped_dists, prob=c(alpha/2))
  q_1_plus_alpha_div_2 = quantile(bootstrapped_dists, prob=c(1-alpha/2))
  CI = c(sample_mean+q_alpha_div_2, sample_mean+q_1_plus_alpha_div_2)
  # Prints result
  if (print_msg) {
    cat(paste("The", (1-alpha)*100, "% bootstrapped confidence interval for the",
              "given data is:\n"))
  }
  # Returns
  CI
}
```

Let’s generate a normal distribution with 10 elements with mean $\mu = 0$ and standard deviation $\sigma = 1$, and compute its 95% bootstrapped confidence interval for the mean with $N_{simulation} = 10000$ as an example:

```
n_simulations = 10000
data = rnorm(10, 0, 1)
mean_CI_bootstrap(data, n_simulations, 0.05, print_msg=T)
```

```
## The 95 % bootstrapped confidence interval for the given data is:
```

```
##       2.5%      97.5%
## -0.6300566  0.2562613
```

It is shown that the real mean $\mu = 0$ falls into the bootstrapped confidence interval for the mean.

Step C

Let's compute a 95% bootstrapped confidence interval for the mean of each period, with $N_{simulation} = 10000$.

```
mean_CI_bootstrap(data_before$firing_rates, n_simulations, 0.05, print_msg=T)
```

```
## The 95 % bootstrapped confidence interval for the given data is:
```

```
##     2.5%    97.5%
## 5.423871 7.531572
```

```
mean_CI_bootstrap(data_during$firing_rates, n_simulations, 0.05, print_msg=T)
```

```
## The 95 % bootstrapped confidence interval for the given data is:
```

```
##     2.5%    97.5%
## 67.61884 85.02273
```

```
mean_CI_bootstrap(data_after$firing_rates, n_simulations, 0.05, print_msg=T)
```

```
## The 95 % bootstrapped confidence interval for the given data is:
```

```
##     2.5%    97.5%
## 7.609258 9.203151
```

The resulting 95% confidence intervals tend to not overlap with each other (Note: in few cases the confidence intervals for the means before and after the puff overlap at their respective upper and lower bounds). this indicates that there is a difference between the three periods in terms of firing rates.

4 - Clustering

Overview

86 neurons of the striatum of a rat performing a spatial task have been recorded. The attached file `action_potential.Rdata` contains the average shape of their action potential as recorded by tetrodes. Typing `load(action_potential.Rdata)` will provide a variable `Record` (a list). `Record[[i]]` is a matrix with 32 lines and 86 columns (one column=one neuron), the index i refers to the electrode number i of the tetrode. The line j in the matrix corresponds to the value of the action potential at $j * 0.0001$ second after the beginning of the action potential.

A – Let's plot for some neurons the superposition of the 4 forms that are corresponding to the action potential as seen by each of the electrodes. It will be shown that the heights of the signal are very differents. This comes from the proximity of the electrodes. IT will also be shown that the highest signal is not always recorded by the same electrode.

B – Let's focus on the best shape, i.e., the shape with the highest peak on all electrodes. On this curve, let's extract two parameters for each neurons, representing the action potential: the duration of the peak above half its height, the delay between the time of the valley after the peak and the time of the peak.

C – Before doing hierarchical clustering, let's center and renormalize both quantities (as when doing PCA) and put the result into a matrix x with 2 columns and 86 rows. Then, let's use `plot(hclust(dist(x),method="ward.D2"))` to see the dendogram. Let's find the largest distance between clusters and cut the tree there. Then let's that there clearly are two clusters.

Of note: *The dendrogram is a visual representation of the data. The two closest points are merged together and represented by their middle (or barycenter) then the algorithm continue and can eventually merge points and barycenters, or two barycenters together until everything is merged. This process of merging is plot as a tree: when two quantities with different heights are merged there are new branches growing and the shortest one correspond to the distance between the centers.*

D – The commands `z=kmeans(x,2)` and `z$cluster` give the cluster label for the different neurons. Let's plot the neurons in the plane of the two variables with a different color code for the first cluster and second cluster as estimated by k-means. this way, "fast spiking interneurons" (the ones with small durations for each delays) and the "medium spiny neurons" have been identified.

Step A

Let's start by importing the data:

```
load("./data/action_potential.Rdata")
action_potentials = Record

cat(paste("Number of electrodes in the tetrode available:", length(action_potentials)))
```

```
## Number of electrodes in the tetrode available: 4
```

```
cat(paste("Number of neurons available:", dim(action_potentials[[1]])[2]))
```

```
## Number of neurons available: 86
```

```
cat(paste("Number of recorded action potential timesteps available per neuron:",
          dim(action_potentials[[1]])[1]))
```

```
## Number of recorded action potential timesteps available per neuron: 32
```

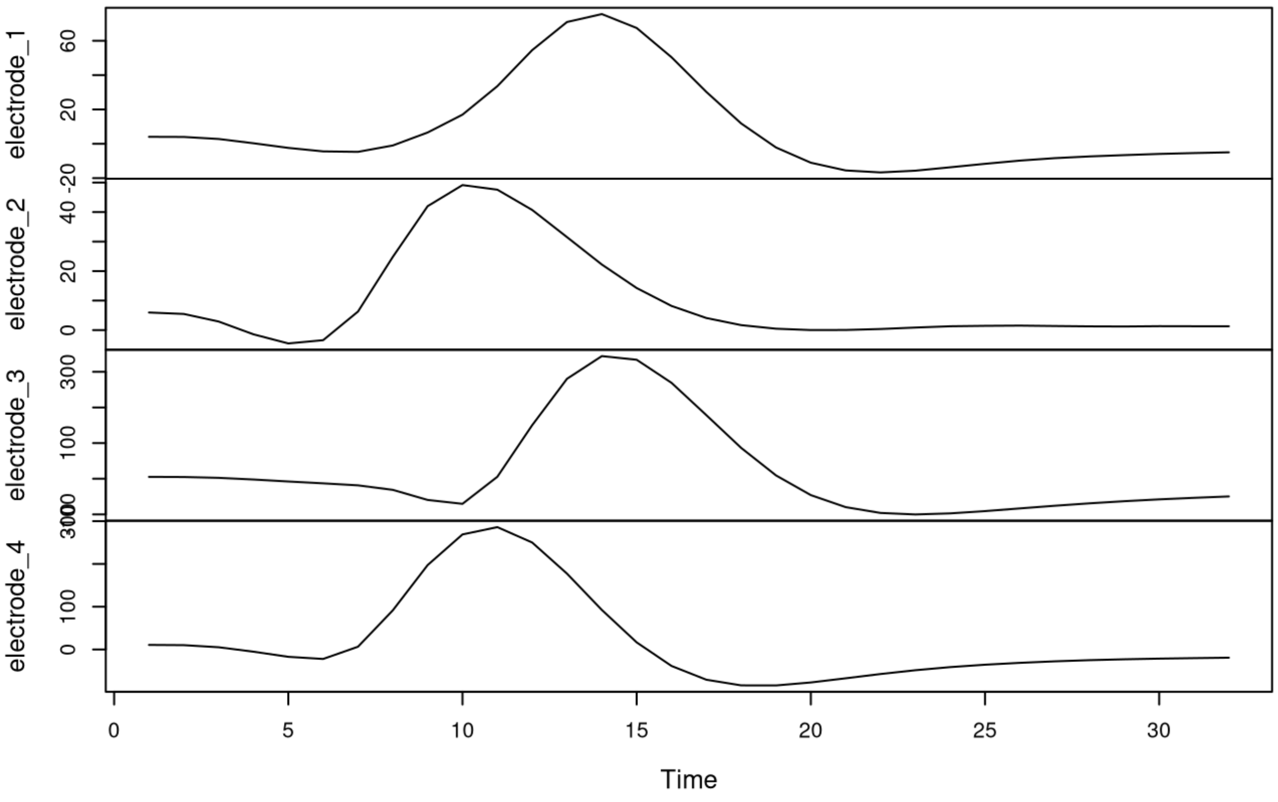
Once the data is imported, let's declare a plotting function to plot selections of neurons from the available data (acquired via each electrode).

```
plot_neuron_data <- function(neuron_number, action_potentials, print_plot=T) {
  # Given a neuron number/index, fetches the corresponding action potential
  # data recorded by each of the four available electrodes and plots the results
  electrode_indexes = matrix(1:4)
  # Retrieves the neuron data recorded by each electrode
  AP_data = apply(electrode_indexes, 1,
                  function(x) {action_potentials[[x]][,neuron_number]})
  AP_data = as.data.frame(AP_data, row.names=c(1:32))
  AP_data = setNames(AP_data, c("electrode_1", "electrode_2", "electrode_3", "electrode_4"))
  # Plots the retrieved data if indicated, if print_plot=F, the function
  # only returns the data
  if (print_plot){
    plot.ts(AP_data,cex.lab=0.8,
            main=paste("Neuron", neuron_number, "'s AP, as recorded",
                       "by the tetrode's 4 electrodes"))
  }
  # Returns the retrieved data
  return(AP_data)
}
```

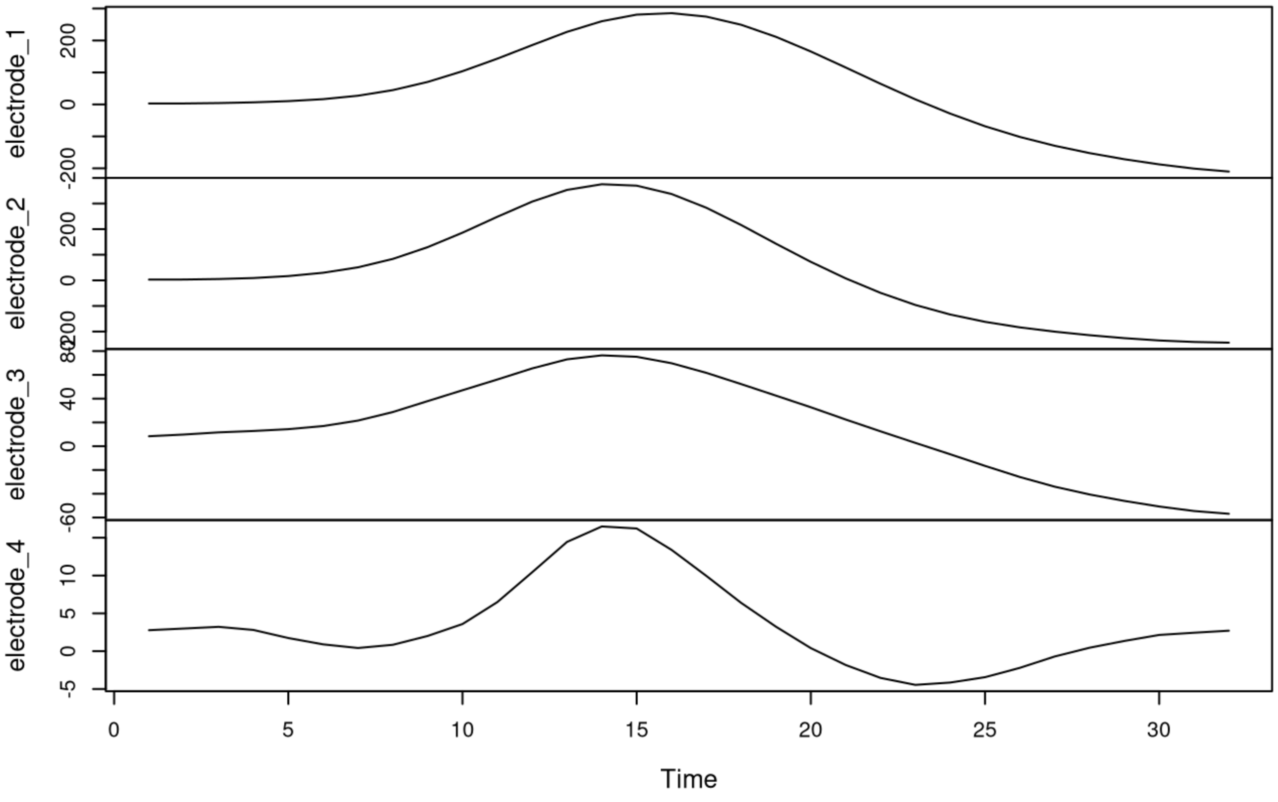
Let's select four random neurons and plot their action potentials with the previously declared functions.

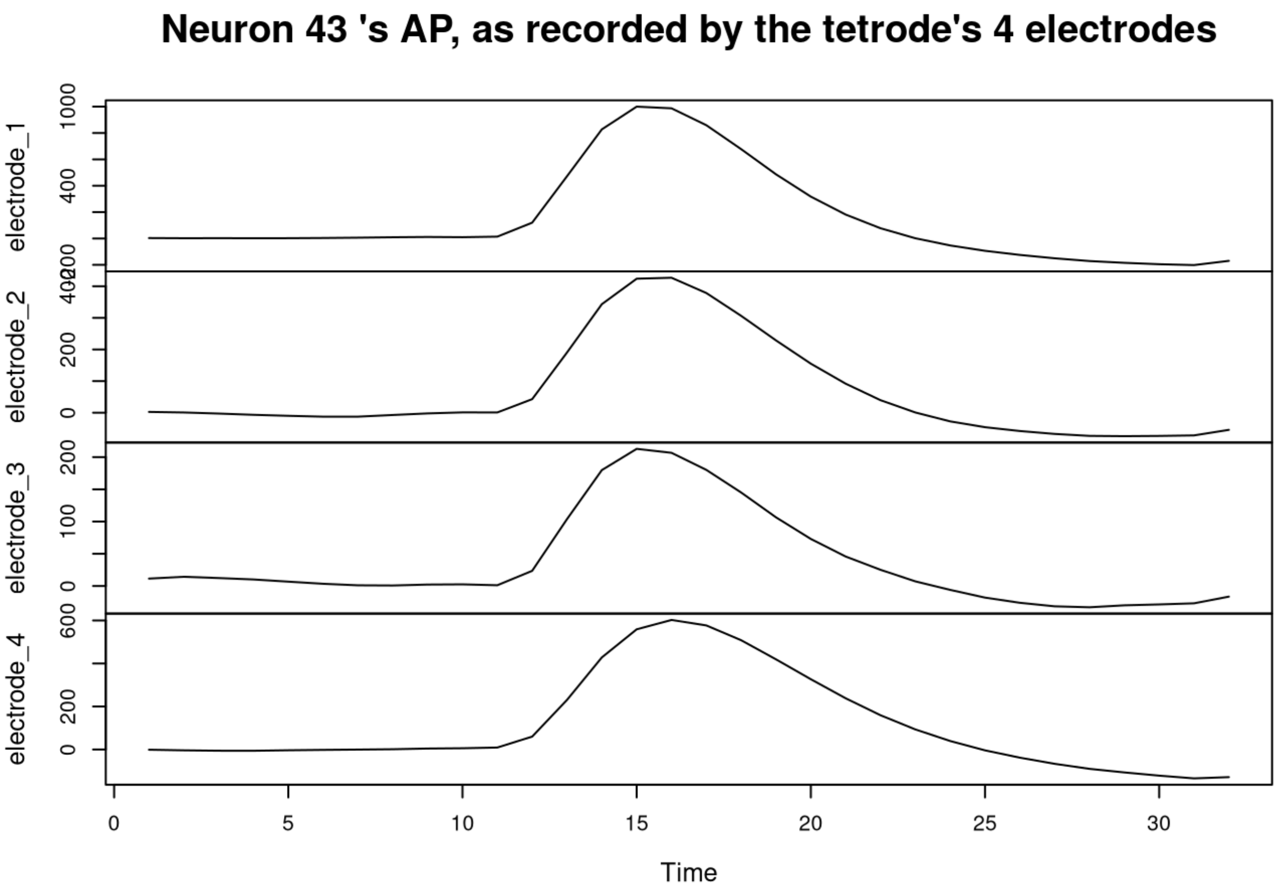
```
neuron_numbers = sample(c(1:dim(action_potentials[[1]])[2]), 4, replace=F)
for (i in neuron_numbers) {
  data = plot_neuron_data(i, action_potentials)
}
```

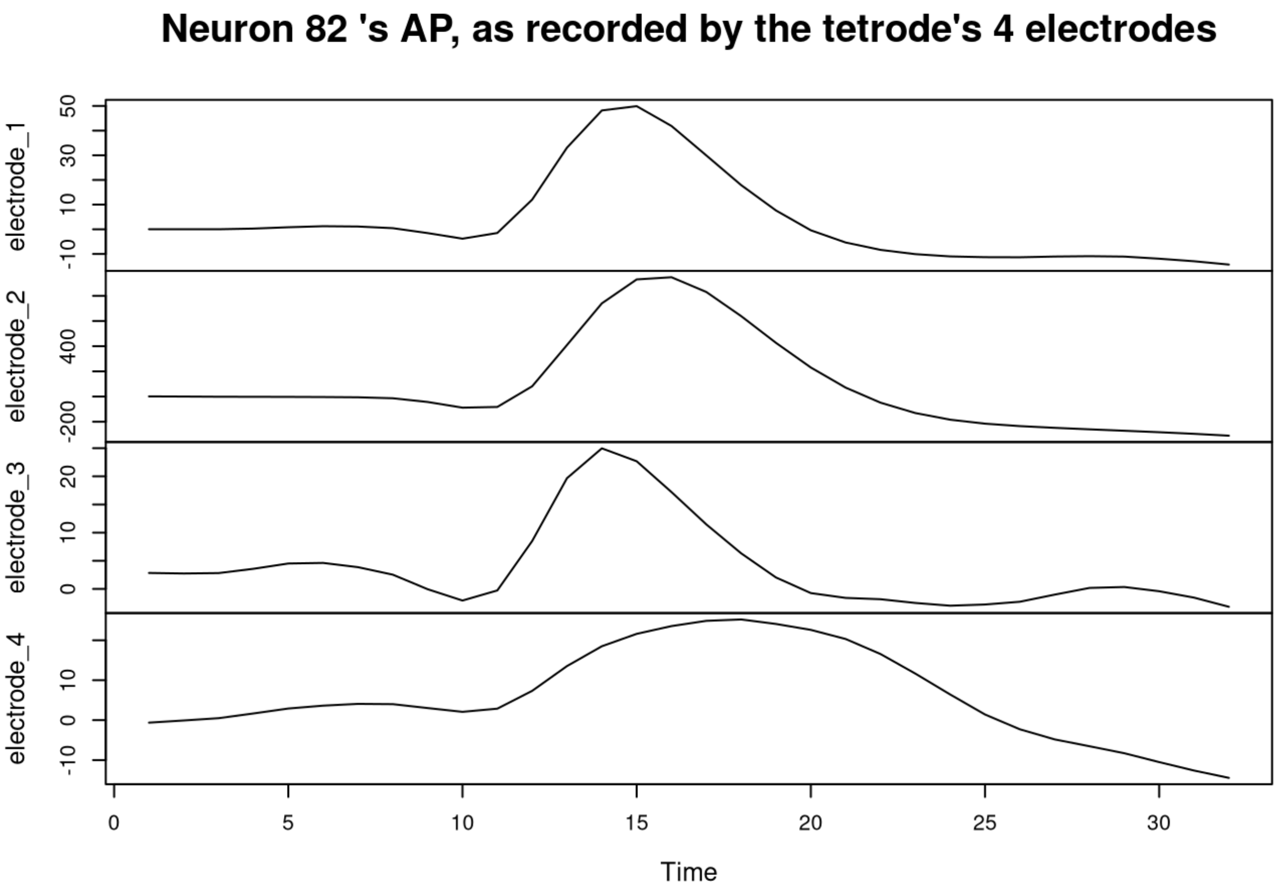
Neuron 57 's AP, as recorded by the tetrode's 4 electrodes



Neuron 19 's AP, as recorded by the tetrode's 4 electrodes







To see which electrode recorded the highest signal for each neuron, let's extract the whole data into a single dataframe.

```
# Retrieves all neurons' AP for all electrodes
full_data = apply(matrix(1:86), 1, function(x) {
                    plot_neuron_data(x, action_potentials, print_plot=F)})

find_electrode_with_highest_signal <- function(full_data){
  # Computes, for each neuron, the electrode that recorded the highest signal
  neuron_indexes = matrix(1:length(full_data))
  apply(neuron_indexes, 1, function(x) {
    round((which(as.matrix(full_data[[x]])==max(full_data[[x]])))/32+1)})
}

# Computes the electrode with the best record per neuron
best_electrodes = find_electrode_with_highest_signal(full_data)
best_electrodes = matrix(best_electrodes, nrow=length(full_data))
row.names(best_electrodes) = c(1:length(full_data))
colnames(best_electrodes) = c("electrode_with_best_record")
```

We now visualize the results:

```
table(best_electrodes)
```

```
## best_electrodes
##  1  2  3  4
## 17 36 27  6
```

In the end, it is found that the electrodes 2 and 3 recorded the highest signal the most, while the electrode 4 recorded the highest signal the least amount of time among the 86 neurons.

Step B

Let's create a function to output the vector $(\text{peak\_to\_valley\_length}, \text{peak\_duration})$ from the best available action potential shape per neuron.

```
parametrize_action_potentials <- function(data, best_electrode){
  # Computes the (peak_to_valley_length,peak_duration) based on the input
  # data and the list of best electrodes computed at the previous step
  #
  # Retrieves data
  best_shape_data = data[,best_electrode]
  # Computes intermediary values
  nb_action_potentials = length(best_shape_data)
  peak_value = max(best_shape_data)
  bottom_value = min(best_shape_data)
  peak_index = which(best_shape_data == peak_value)
  pre_peak_data = best_shape_data[1:(peak_index)] # note:
  #   we include the peak point if the jump is too sharp
  post_peak_data = best_shape_data[(peak_index+1):nb_action_potentials]
  next_bottom_value = min(post_peak_data)
  next_bottom_index = peak_index + which(post_peak_data == next_bottom_value)
  half_height = (peak_value - bottom_value)/2
  pre_peak_half_height_index = which(pre_peak_data>half_height)[1]
  post_peak_half_height_index = which(post_peak_data<half_height)[1]
  # Computes output value
  peak_to_valley_length = next_bottom_index-peak_index
  peak_duration = (peak_index + post_peak_half_height_index) - pre_peak_half_height_index
  # Returns result
  return(list("peak_to_valley_length"=peak_to_valley_length,
              "peak_duration"=peak_duration))
}

parameters = apply(matrix(1:length(full_data)), 1,
                   function(x){parametrize_action_potentials(
                     full_data[[x]],
                     best_electrodes[x]
                   )})
```

Let’s display the first five results:

```
for (neuron in 1:5){
  cat("Case: Neuron", neuron,"\n")
  cat("The peak to valley duration was: c.",
      parameters[[neuron]]$peak_to_valley_length * 0.0001,
      "seconds.\n")
  cat("The peak duration was: c.",
      parameters[[neuron]]$peak_duration * 0.0001,
      "seconds.\n\n")
}
```

```
## Case: Neuron 1
## The peak to valley duration was: c. 0.0018 seconds.
## The peak duration was: c. 6e-04 seconds.
##
## Case: Neuron 2
## The peak to valley duration was: c. 0.0018 seconds.
## The peak duration was: c. 6e-04 seconds.
##
## Case: Neuron 3
## The peak to valley duration was: c. 0.0017 seconds.
## The peak duration was: c. 5e-04 seconds.
##
## Case: Neuron 4
## The peak to valley duration was: c. 0.0016 seconds.
## The peak duration was: c. 4e-04 seconds.
##
## Case: Neuron 5
## The peak to valley duration was: c. 0.0017 seconds.
## The peak duration was: c. 4e-04 seconds.
```

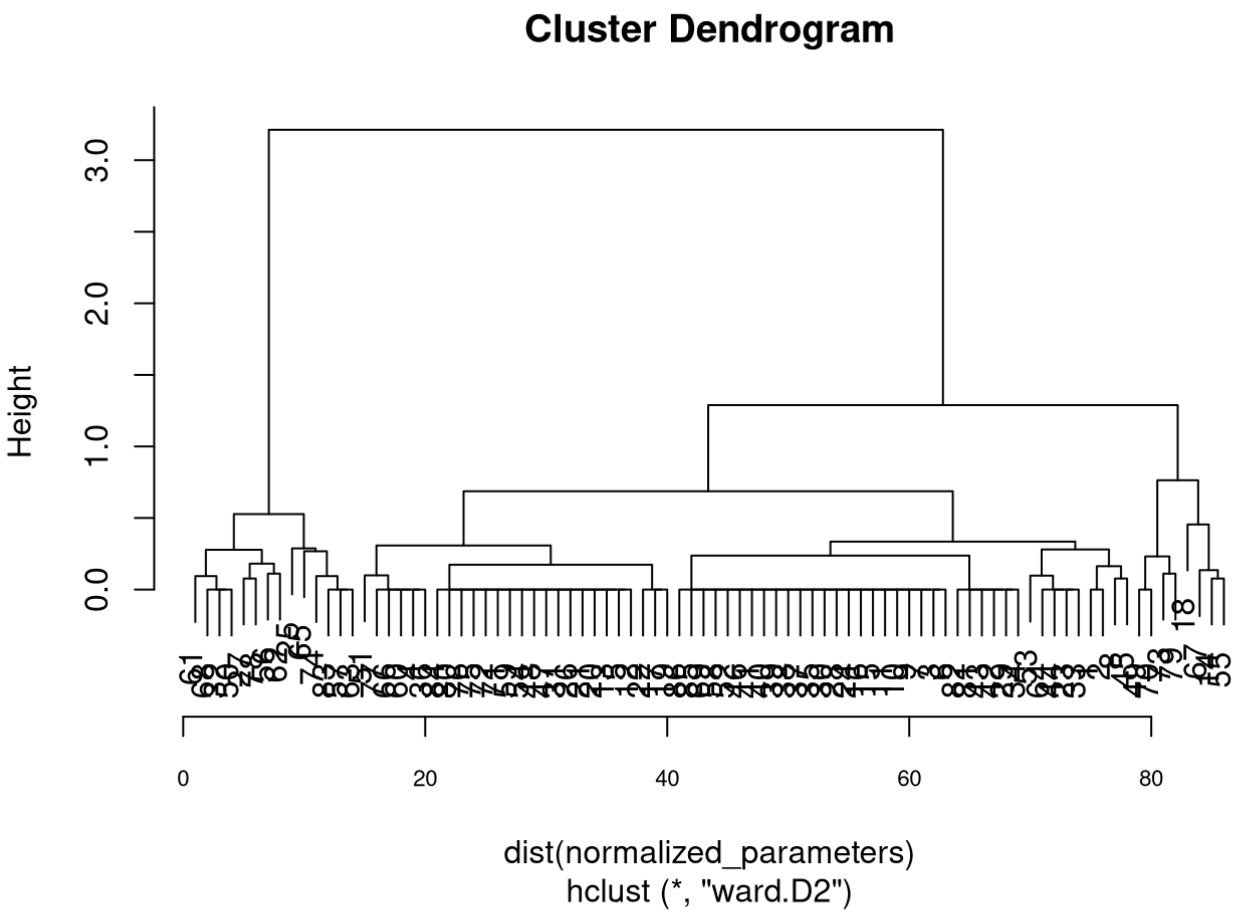
Step C

Let’s start by reformatting our data into a 86×2 matrix and standardizing its content:

```
parameters = t(matrix(unlist(parameters),nrow=2,ncol=86))
row.names(parameters) = c(1:length(full_data))
colnames(parameters) = c("peak_to_valley_length", "peak_duration")
standardize <- function(x){(x-min(x))/(max(x)-min(x))}
normalized_parameters = apply(parameters, 2, standardize)
```

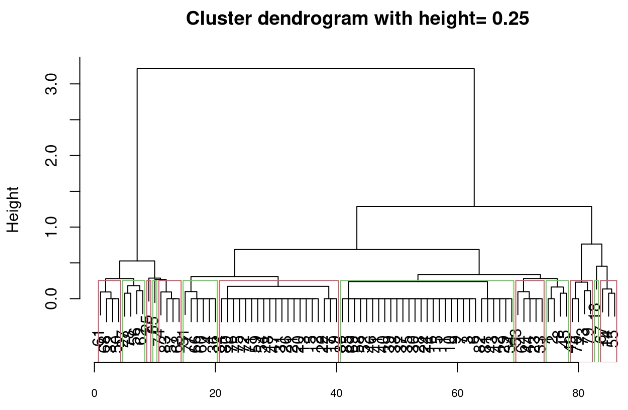
Let’s now compute the dendrogram of the centered and normalized set of data:

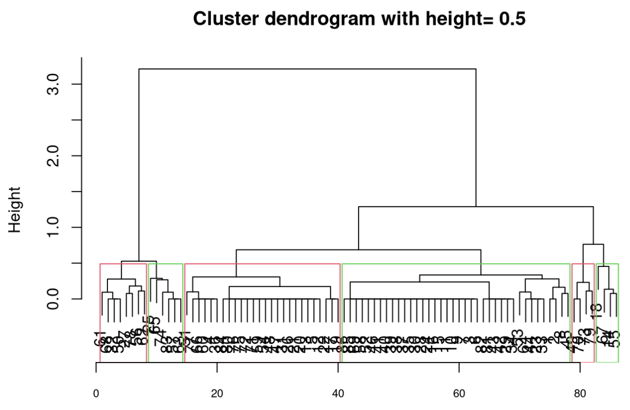
```
clustering = hclust(dist(normalized_parameters),method="ward.D2")
plot(clustering)
axis(1, cex.axis = 0.7)
```

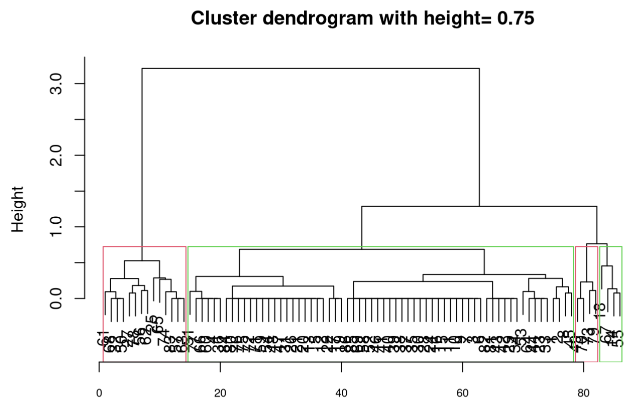


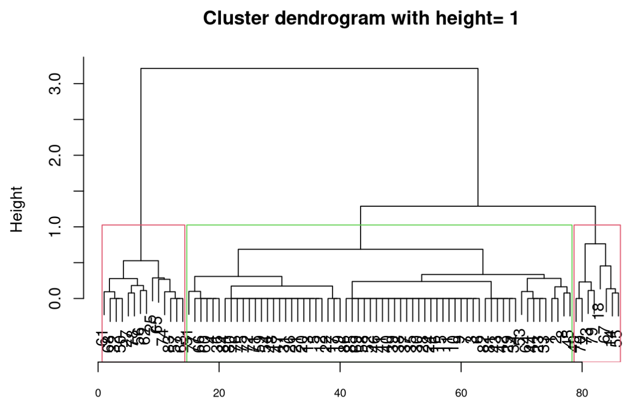
To find the largest distance between clusters and cut the tree there, let's iterate over the sequence of heights $\{0.25, 0.5, \ldots, 2.75\}$ and then try with an infinite height as a clsutering parameter. Then, let's visualize the resulting dendrogram clustering. It is found that the larest distance between clusters is achieved once height reaches a value between 1.25 and 1.5.

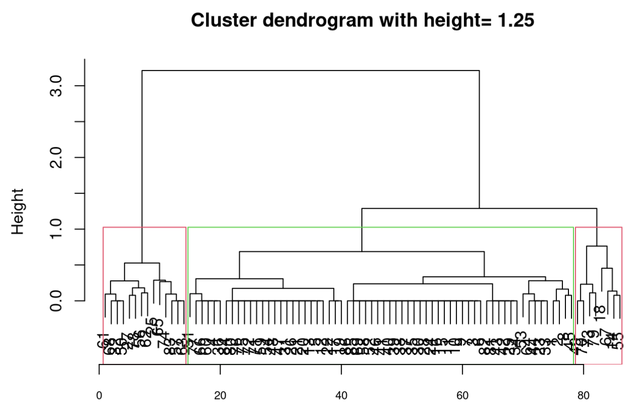
```
for (i in c(seq(0.25,2.75, 0.25),Inf)){
  plot(clustering, xlab="",main=paste("Cluster dendrogram with height=",i),sub="")
  rect.hclust(clustering,h = i,border=c(2:3))
  axis(1, cex.axis = 0.7)
}
```

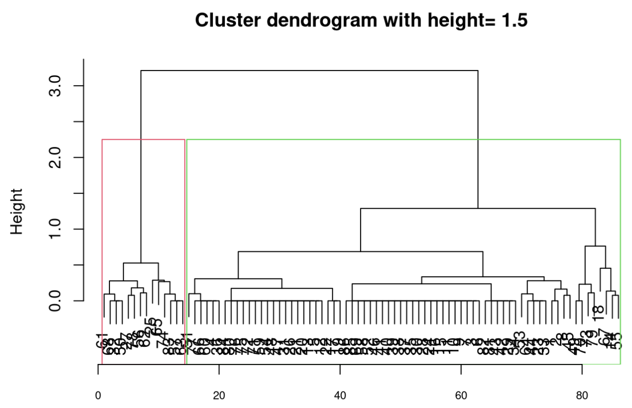


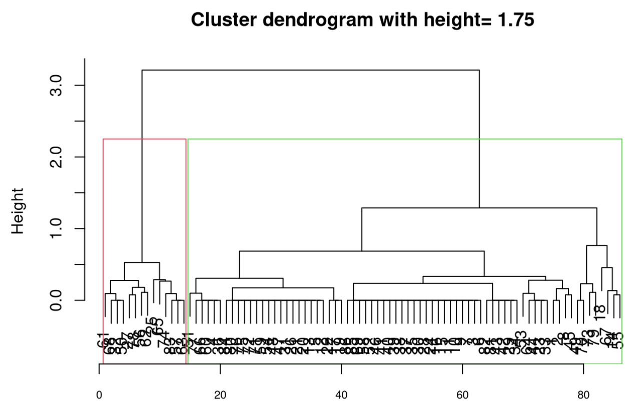


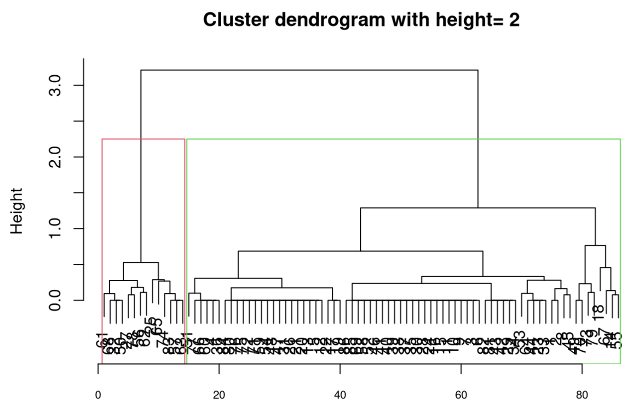

Cluster dendrogram with height= 0.75
Height
0.0
1.0
2.0
3.0
0
20
40
60
80

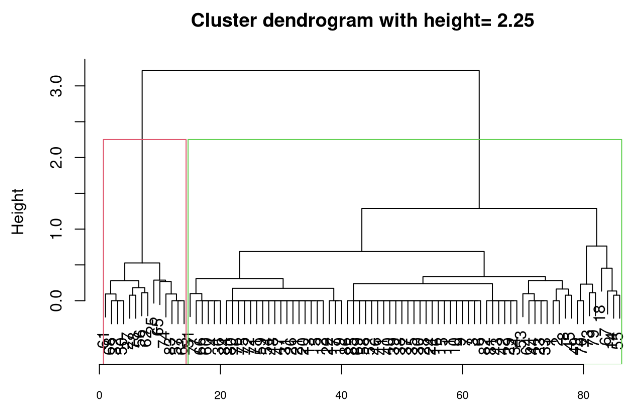

Cluster dendrogram with height= 1
Height
0.0
1.0
2.0
3.0
0
20
40
60
80

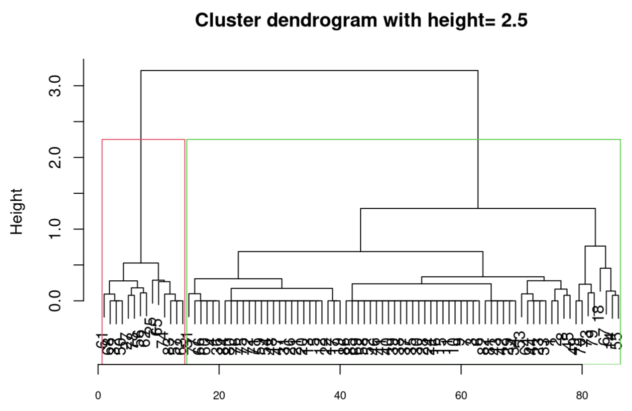

Cluster dendrogram with height= 1.25
Height
0.0
1.0
2.0
3.0
0
20
40
60
80

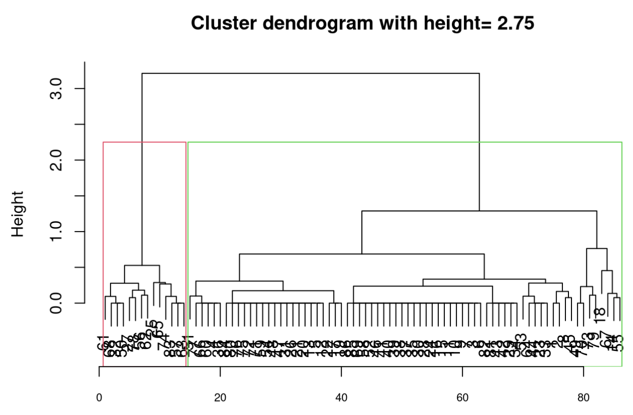

Cluster dendrogram with height= 1.5
Height
0.0
1.0
2.0
3.0
0
20
40
60
80

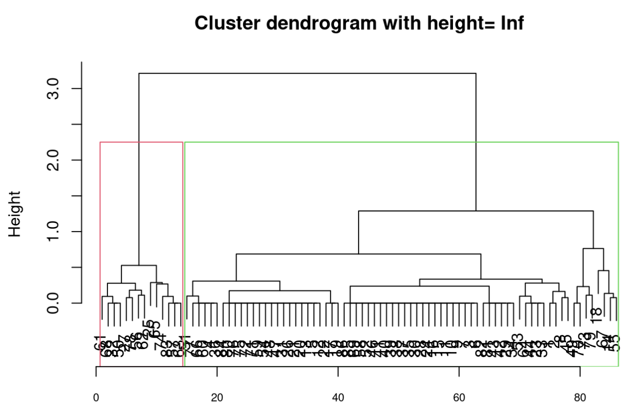

Cluster dendrogram with height= 1.75
Height
0.0
1.0
2.0
3.0
0
20
40
60
80


Cluster dendrogram with height= 2
Height
0.0
1.0
2.0
3.0
0
20
40
60
80


Cluster dendrogram with height= 2.25
Height
0.0
1.0
2.0
3.0
0
20
40
60
80

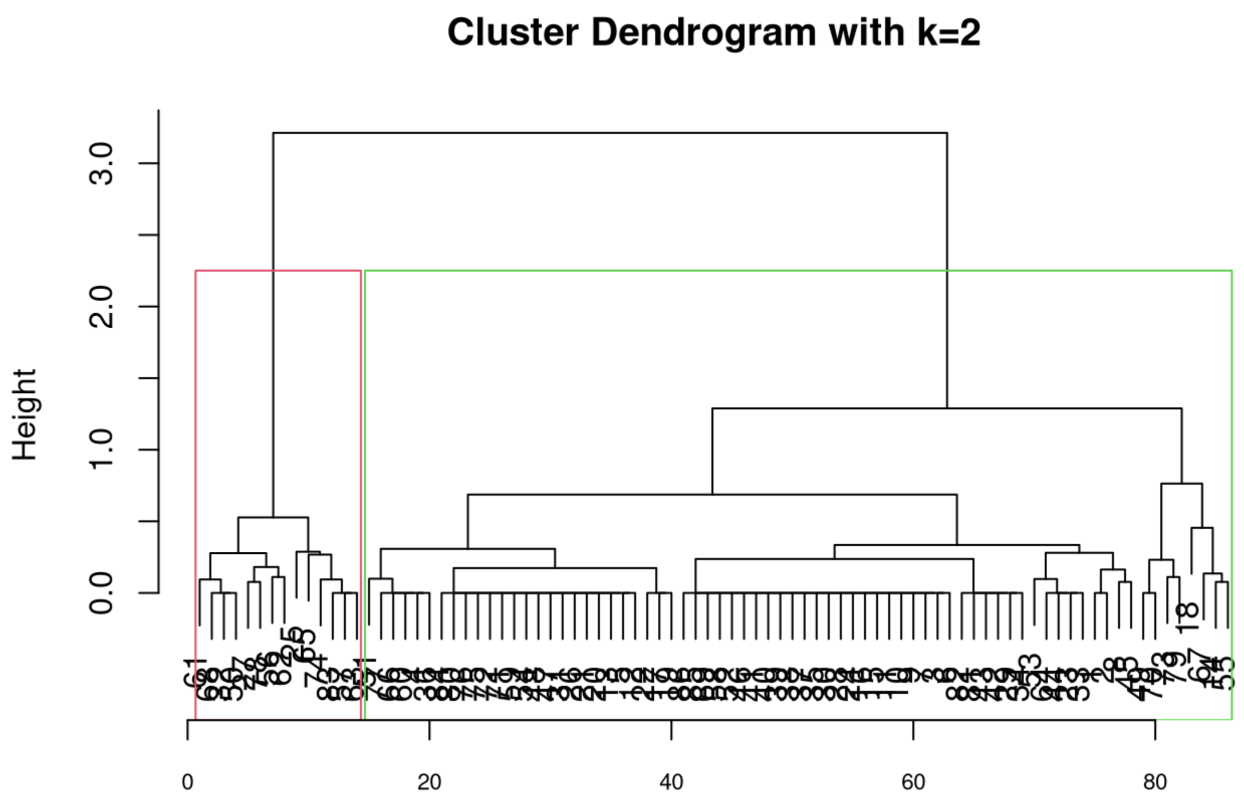

Cluster dendrogram with height= 2.5
Height
0.0
1.0
2.0
3.0
0
20
40
60
80





As such, it can confidently be said that two distinct clusters of action potentials are found such that:

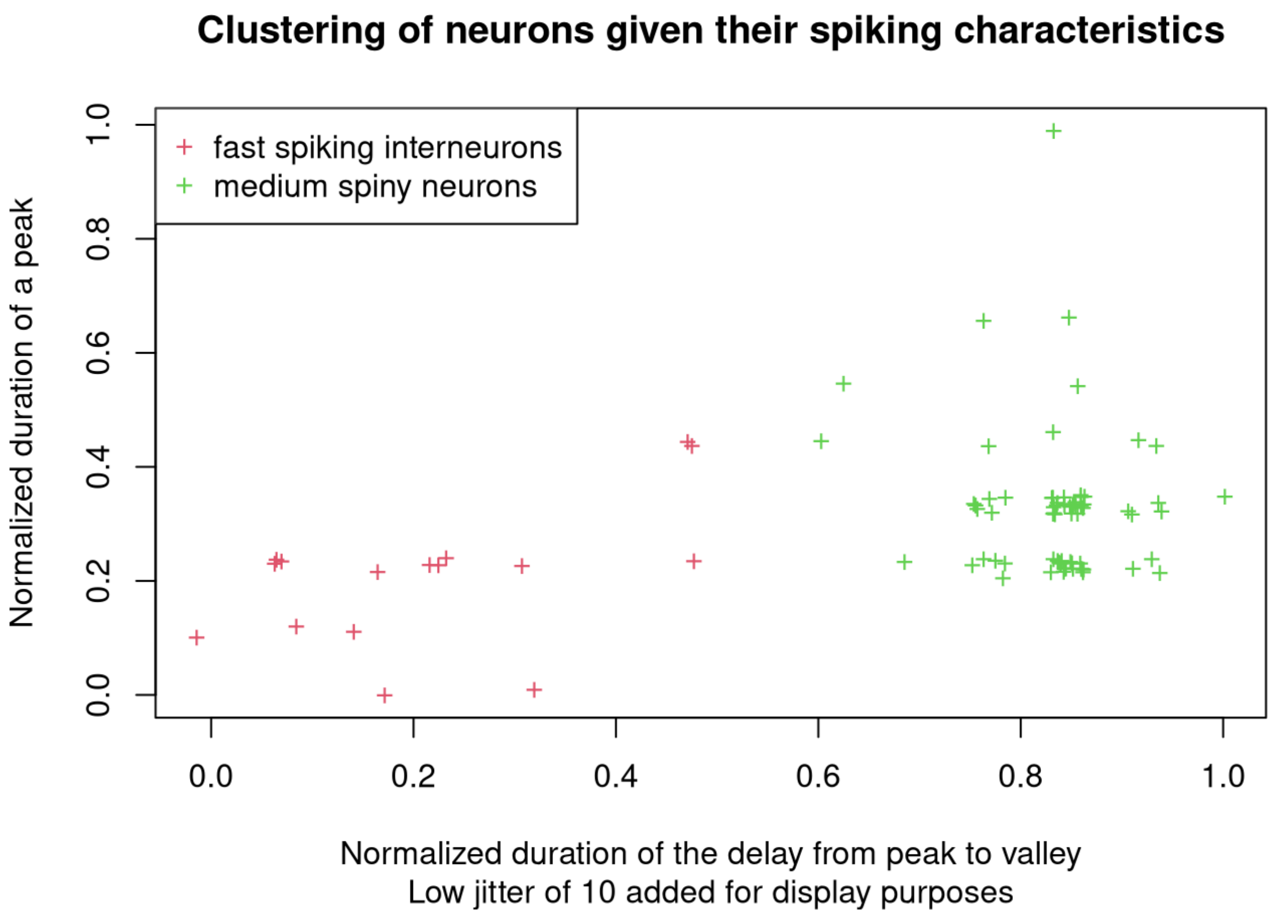
```
plot(clustering, xlab="", sub="",
     main= "Cluster Dendrogram with k=2")
rect.hclust(clustering,k=2,border=c(2:3))
axis(1, cex.axis = 0.7)
```



Step D

Let’s plot the neurons in the plane of the two previously declared variables with a different color code for each cluster (as estimated by a k-means method). This way, “fast spiking interneurons” (i.e. neurons with small durations for each delays) and the “medium spiny neurons” can easily be identified:

```r
# Computes clustering with kmeans
z = kmeans(normalized_parameters, 2)

# Determines the color order for FSI
if (z$cluster[unique(which(normalized_parameters[,1]==min(normalized_parameters[,1])))][1]==1){
  colors = c(2,3)
} else {
  colors = c(3,2)
}

# Plots
plot(jitter(normalized_parameters,10), col=z$cluster+1,
     xlab="Normalized duration of the delay from peak to valley",
     ylab="Normalized duration of a peak",
     main="Clustering of neurons given their spiking characteristics",
     sub="Low jitter of 10 added for display purposes", pch="+")
legend("topleft",col=colors,
       legend=c("fast spiking interneurons", "medium spiny neurons"),
       pch="+")
```


Clustering of neurons given their spiking characteristics
fast spiking interneurons
medium spiny neurons
Normalized duration of a peak
Normalized duration of the delay from peak to valley
Low jitter of 10 added for display purposes