Density estimation

Computational Statistics

Adam Gorm Hoffmann

Goal

Compute kernel density estimate

$$\hat{f}_h(x) = \frac{1}{hn}\sum_{j=1}^{n} K\left(\frac{x - x_j}{h}\right)$$

where

$$K(x) = \frac{3}{4}(1 - x^2)1_{[-1,1]}(x)$$

is the Epanechnikov kernel.

- First: How and where to compute the kernel density estimate?
  - Basic implementation.
  - Different binning implementations (R and Rcpp).
- Then (no time. . .): How to choose a reasonable bandwidth h?
  - k-fold cross validation.
  - AMISE plug-in.

Where to estimate density?

An extreme case would be to estimate the density in all $x_1, \dots, x_n$:

$$\hat{f}_h(x_i) = \frac{1}{hn} \sum_{j=1}^{n} K\left(\frac{x_i - x_j}{h}\right)$$

which requires n^2 kernel evaluations ($\frac{1}{2}(n-1)^2 + 1$ when using symmetry of K).

Instead we only estimate it in grid points $g_1, \dots, g_m$:

$$\hat{f}_h(g_i) = \frac{1}{hn} \sum_{j=1}^{n} K\left(\frac{g_i - x_j}{h}\right)$$

which requires (half of) mn kernel evaluations.

C++ functions and R wrapper

```cpp
double
ep_density_single_cpp(double x, const NumericVector& x_obs, double h)
{
  double result = 0;
  unsigned long long n = x_obs.length();
  for (unsigned long long i = 0; i < n; i++)
    result += ep_kernel_single_cpp((x - x_obs[i]) / h);
  return result / (h * n);
}

// [[Rcpp::export]]
NumericVector
ep_density_cpp(const NumericVector& x_obs, double h, const NumericVector& grid)
{
  NumericVector density_estimates(grid.length());
  for (int i = 0; i < grid.length(); i++)
    density_estimates[i] = ep_density_single_cpp(grid[i], x_obs, h);
  return density_estimates;
}
```

R wrapper:

```r
ep_density_cpp_wrap <- function(x_obs, h, m = 512)
{
    grid <- seq(min(x_obs) - 3 * h, max(x_obs) + 3 * h, length.out = m)
    list(
        x = grid,
        y = ep_density_cpp(x_obs, h, grid)
    )
}
```

Binning: further reducing kernel evaluations

Assign each grid point g_j the grid count

$$w_j = \sum_{i=1}^{n} 1(g_j \text{ is the grid point closest to } x_i)$$

and use the kernel density estimate

$$\hat{f}_h(g_i) = \frac{1}{hn} \sum_{j=1}^{m} w_j \cdot K\left(\frac{g_i - g_j}{h}\right)$$

which requires only (half of) m^2 kernel evaluations.

Less kernel evaluations at cost of accuracy.

Even fewer evaluations with equidistant grid

Note that when $g_1, \ldots, g_m$ are equidistant as

$$g_1, g_1 + \delta, g_1 + 2\delta, \ldots, g_1 + (m-1)\delta$$

then

$$|g_i - g_j| \in \{0, \delta, \ldots, (m-1)\delta\}, \quad \text{for all } i, j = 1, \ldots, m$$

so we only have to do m kernel evaluations

$$K\left(\frac{0}{h}\right), K\left(\frac{\delta}{h}\right), \ldots, K\left(\frac{(m-1)\delta}{h}\right)$$

when we also use that K is symmetric.

And even fewer for kernels where $K(y) = 0$ for $|y| \geq 1$

Note that

$$K\left(\frac{i\delta}{h}\right) = 0, \quad \text{for } i \geq \frac{h}{\delta}$$

so we only have to calculate

$$K\left(0\right), K\left(\frac{\delta}{h}\right), \ldots, K\left(\texttt{floor}\left(\frac{h}{\delta}\right)\frac{\delta}{h}\right)$$

which amounts to `ceiling` $\left(\frac{h}{\delta}\right)$ kernel evaluations.

Example: Standard Gaussian

We choose the following reasonable parameters:

- $n = 10^5$ observations.
- $m = 512$ grid points in $[-3\sigma, 3\sigma] = [-3, 3]$.
- $\delta = \frac{3-(-3)}{512} \approx 0.01$.
- $h = 1.06\sigma n^{-1/5} \approx 0.1$.

Number of kernel evaluations	Method
$n^2 = 10^{10}$	evaluating in all observations
$mn = 512 \cdot 10^5 = 5.12 \cdot 10^7$	evaluating in grid
$m^2 \approx 2.6 \cdot 10^5$	naive binning
$m = 512$	naive equidistant binning
$\texttt{ceiling}\left(\frac{h}{\delta}\right) = 10$	smart equidistant binning

We save a lot of kernel evaluations! But we still need some iterations to sum the weighted kernel values.

R implementation of weights

```r
bin_weights <- function(x, grid_lower, grid_length, grid_diff)
{
  w <- numeric(grid_length)
  for(i in seq_along(x)) {
    closest_grid_i <- ceiling((x[i] - grid_lower) / grid_diff + 0.5)
    w[closest_grid_i] <- w[closest_grid_i] + 1
  }
  w
}
```

- Very easy to translate to C++.

Rcpp implementation of weights

```cpp
NumericVector
bin_weights_cpp(const NumericVector& x, double grid_lower,
                int grid_length, double grid_diff)
{
  double closest_grid_i;
  NumericVector w(grid_length);
  unsigned long long n_x = x.length();

  for (unsigned long long i = 0; i < n_x; i++) {
    closest_grid_i = floor((x[i] - grid_lower) / grid_diff + 0.5);
    w[closest_grid_i] += 1;
  }

  return w;
}
```

R implementation of binning

```r
ep_density_binning <- function(x, h, grid_length = 512)
{
    grid_lower <- min(x) - 3 * h
    grid_upper <- max(x) + 3 * h
    grid <- seq(grid_lower, grid_upper, length.out = grid_length)
    grid_diff <- grid[2] - grid[1]
    max_nonzero_i <- floor(h / grid_diff)
    kernel_evals <- ep_kernel_cpp((grid[1:(max_nonzero_i + 1)] -
                                   grid_lower) / h)
    kernel_vec <- c(rev(kernel_evals[-1]), kernel_evals)
    weights <- c(rep(0, max_nonzero_i),
                 bin_weights_cpp(x, grid_lower, grid_length, grid_diff),
                 rep(0, max_nonzero_i))
    y <- numeric(grid_length)
    for (i in (1 + max_nonzero_i):(grid_length + max_nonzero_i))
        y[i - max_nonzero_i] <- sum(weights[(i - max_nonzero_i):
                                            (i + max_nonzero_i)] *
                                    kernel_vec)

    list(x = grid, y = y / (h * length(x)))
}
```

R implementation with Toeplitz trick

The kernel is evaluated in all grid points, but all computations are vectorized.

```r
ep_density_binning_toeplitz <- function(x, h, grid_length = 512)
{
    n <- length(x)
    grid_lower <- min(x) - 3 * h
    grid_upper <- max(x) + 3 * h
    grid <- seq(grid_lower, grid_upper, length.out = grid_length)
    grid_diff <- grid[2] - grid[1]
    kernel_evals <- ep_kernel_cpp((grid - grid_lower) / h)
    weights <- bin_weights_cpp(x, grid_lower, grid_length, grid_diff)
    list(
        x = grid,
        y = colSums(weights * toeplitz(kernel_evals)) / (h * length(x))
    )
}
```

Problems with profiling the implementations

- They are too fast to be profiled.
- When profiling a replication of them (so they take longer), the results vary wildly.

Solution

- Benchmark each line manually with `bench::mark`
- I do it for 10^8 observations x.
- Possible package idea? Seems useful!

Benchmarking: `bin_weights`

R implementation: 16.3s

Rcpp implementation: 450ms

As expected, since we didn't vectorize anything in R.

Benchmarking: `min` and `max` are slow! So is `range`.

Expression	Median time (ms)
`c(min(x), max(x))`	259
`range(x)`	715
`range_cpp(x)`	97

Solution: I wrote my own fast `range` copy in C++.

```cpp
NumericVector
range_cpp(const NumericVector& x)
{
  unsigned long long n_x = x.length();
  NumericVector range(2);
  range[0] = x[0];
  range[1] = x[0];
  for (unsigned long long i = 1; i < n_x; i++) {
    if (x[i] < range[0])
      range[0] = x[i];
    else if (x[i] > range[1])
      range[1] = x[i];
  }
  return range;
}
```

Benchmark-profiling the smart implementation

Kernel evaluations: 0.002ms.

`bin_weights_cpp`: 458ms.

The loop: 5ms.

Bottlenecks: `bin_weights_cpp`, `min` and `max`.

- `min` and `max` can't easily be improved more than `range_cpp`.
    - Possible solution is to let the user supply end-points of grid manually, if they have prior knowledge of a reasonable grid.
- `bin_weights_cpp` seems to be optimized in C++.
- Note: This is for very a very large data set (10^8 obs.)
- When the number of observations is not much larger than the grid length, the loop and kernel evaluations matter more (so it still makes sense to optimize them in C++, as we will see).

Benchmark-profiling Toeplitz implementation

Kernel evaluations: 0.005ms.

`bin_weights_cpp`: 458ms.

`toeplitz(kernel_evals)`: 6ms.

Multiplying with `weights`: 1ms.

`colSums`: 0.2ms.

- `weights`, `min`, and `max` are still the largest bottlenecks.
- The Toeplitz implementation is a few ms slower than the smart R implementation – the grid length is so little compared to the length of `x`, that doing 512 kernel evaluations instead of 4 doesn't really matter.

Conclusions from benchmark-profiling

- Lines that scale with the size of `x` are the bottlenecks (`bin_weights`, `min`, `max`).
- We have reduced their runtime significantly with C++ implementations, but they are still the bottlenecks.
- We will now do a complete C++ implementation, and see that it still saves time for smaller sample sizes.

Rcpp implementation: highest level

```cpp
NumericVector
ep_density_binning_cpp(const NumericVector& x, double h,
                       const NumericVector& grid)
{
  int i;
  int n_x = x.length();
  int grid_length = grid.length();
  double grid_lower = grid[0];
  double grid_diff = grid[1] - grid[0];
  NumericVector y(grid_length);
  NumericVector weights = bin_weights_cpp(x, grid_lower,
                                          grid_length, grid_diff);
  int max_nonzero_i = floor(h / grid_diff);
  double kernel_evals[max_nonzero_i + 1];
  kernel_evals[0] = 0.75; // Ep kernel in 0.
  for (i = 1; i <= max_nonzero_i; i++)
    kernel_evals[i] = ep_kernel_single_cpp((i * grid_diff) / h);
  for (i = 0; i < grid_length; i++)
    y[i] = weighted_kernel_sum(i, max_nonzero_i, grid_length,
                               weights, kernel_evals) / (h * n_x);
  return y;
}
```

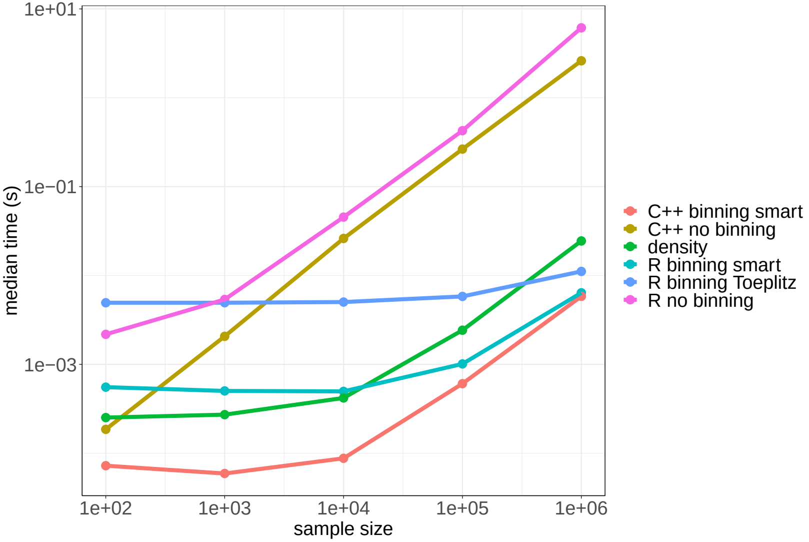
Rcpp implementation: weighted kernel sum

```cpp
double
weighted_kernel_sum(double i, double max_nonzero_i, double grid_length,
                    const NumericVector& weights,
                    const double *kernel_evals)
{
  int below, above;
  double res = weights[i] * *kernel_evals;
  for (int j = 1; j <= max_nonzero_i; j++) {
    below = i - j;
    above = i + j;
    if (below >= 0)
      res += weights[below] * *(kernel_evals + j);
    if (above < grid_length)
      res += weights[above] * *(kernel_evals + j);
  }

  return res;
}
```

Rcpp R wrapper

```r
ep_density_binning_cpp_wrap <- function(x, h, grid_length = 512)
{
    grid <- seq(min(x) - 3 * h, max(x) + 3 * h, length.out = grid_length)
    list(
        x = grid,
        y = ep_density_binning_cpp(x, h, grid)
    )
}
```

Benchmarking on Gaussian mixture



- As expected, C++ doesn't make much difference for very large samples.

Finally: my own `density` copy

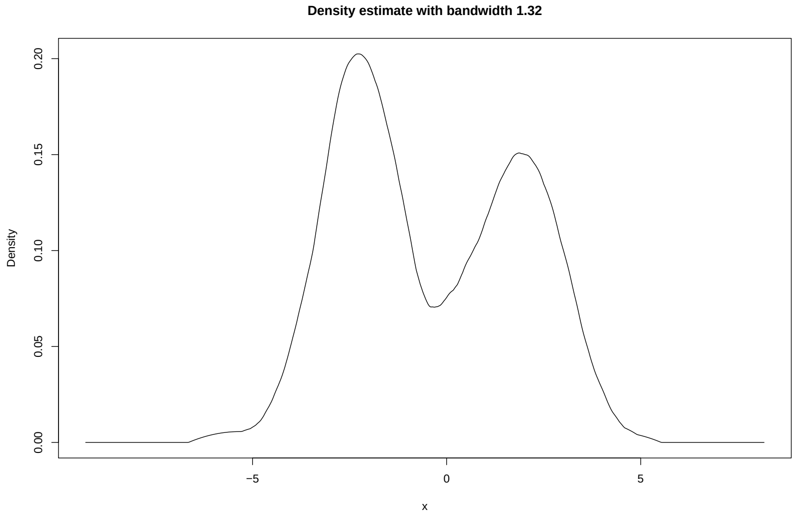
```r
my_density <- function(x, bw = "plug-in", binning = TRUE, ...)
{
    if (bw == "plug-in")
        h <- bandwidth_plug_in_precalculated(x)
    else if (bw == "cv")
        h <- bandwidth_cv(x, ...)
    else if (bw == "silverman")
        h <- bandwidth_silverman(x)
    else if (is.function(bw))
        h <- bw(x, ...)
    else if (is.numeric(bw))
        h <- bw
    else
        stop("Invalid bw.")

    if (binning)
        xy <- ep_density_binning_cpp_wrap(x, h, ...)
    else
        xy <- ep_density_cpp_wrap(x, h, ...)

    structure(
        list(
            x = xy$x,
            y = xy$y,
            bw = h
        ),
        class = "my_density"
    )
}
```

Plot method

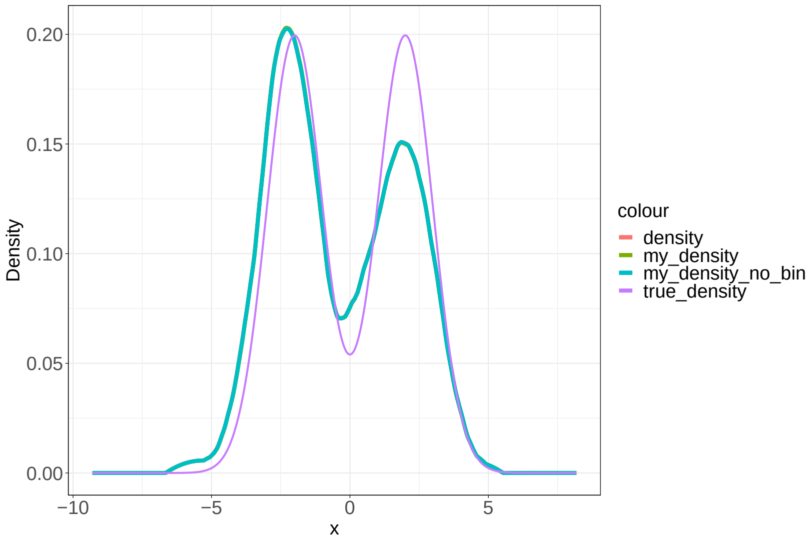
```r
plot.my_density <- function(my_density_object, ...)
    plot(my_density_object$x, my_density_object$y,
         type = "l", xlab = "x", ylab = "Density",
         main = paste("Density estimate with bandwidth",
                      round(my_density_object$bw, digits = 2)),
         ...)
```

Testing on Gaussian mixture: `plot` method.

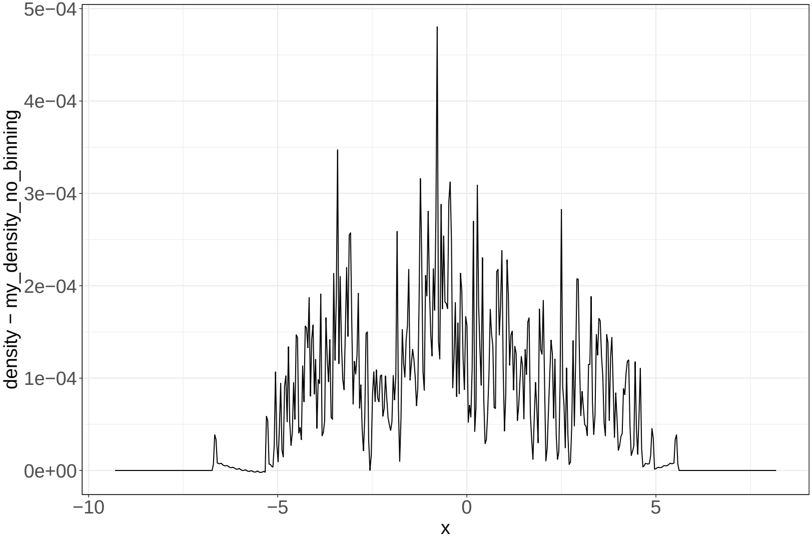
```r
plot(my_density(x, bw = "plug-in", binning = FALSE))
```



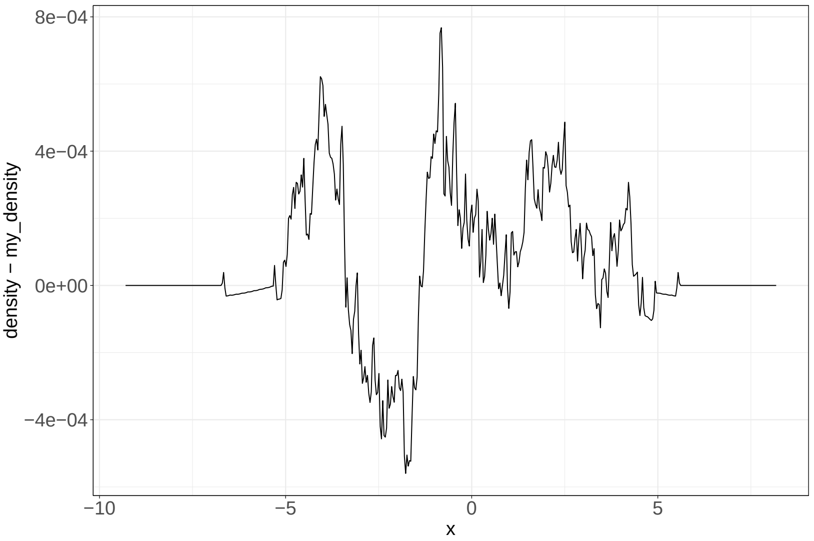
my_density and density with same bandwidth



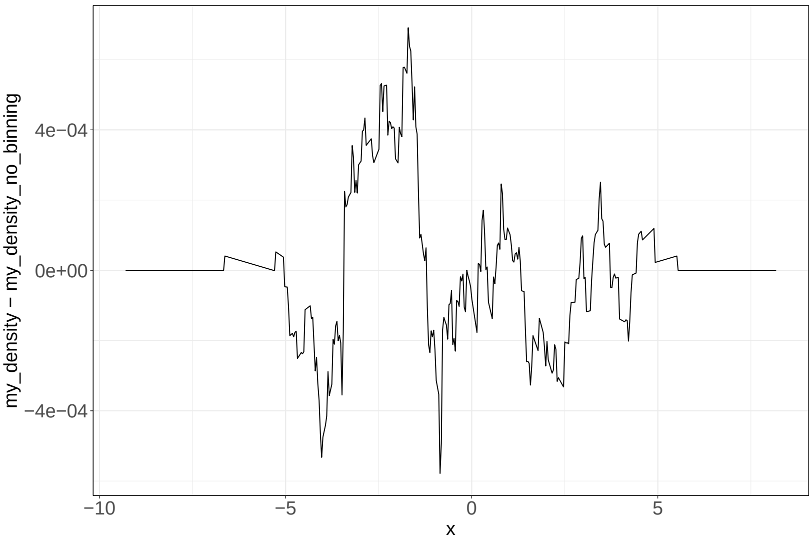
Differences between `density` and non-binning imp.



Differences between `density` and binning imp.

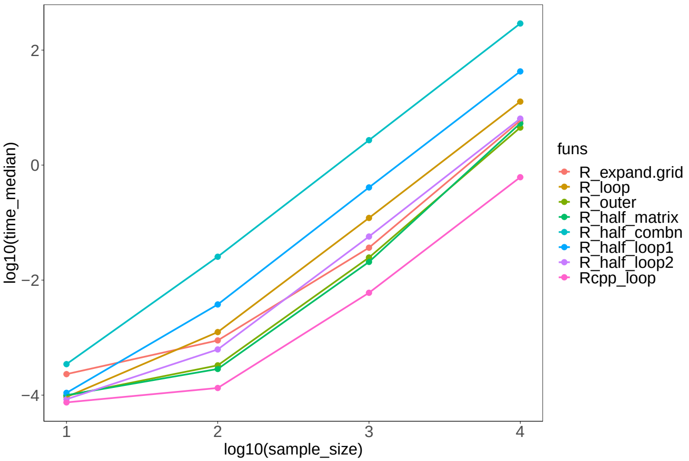


Differences between binning and non-binning imp.



Bandwidth selection with k-fold cross-validation and plug-in

- k-fold cross validation
  - Improve speed by caching complements of folds I^{-i}.
- Plug-in
  - Improve speed with Rcpp implementation.



k-fold cross-validation

Select bandwidth

$$\hat{h}_{\mathrm{CV}(k)} = \arg\,\max_h \sum_{i=1}^{n} \log(\hat{f}_h^{-i}).$$

where

$$\hat{f}_h^{-i} = \frac{1}{hn_i} \sum_{j \in I^{-i}} K\left(\frac{x_i - x_j}{h}\right)$$

is the kernel density estimate in x_i based on the $k-1$ folds that don't contain i.

Highest level of implementation

```r
## Selects h by k-fold cross validation
bandwidth_cv <- function(x_obs, k)
{
    optimize(
        f = get_cv_objective(x_obs, k),
        interval = c(0, max(x_obs) - min(x_obs)),
        maximum = TRUE
    )$maximum
}
```

Naive objective function implementation

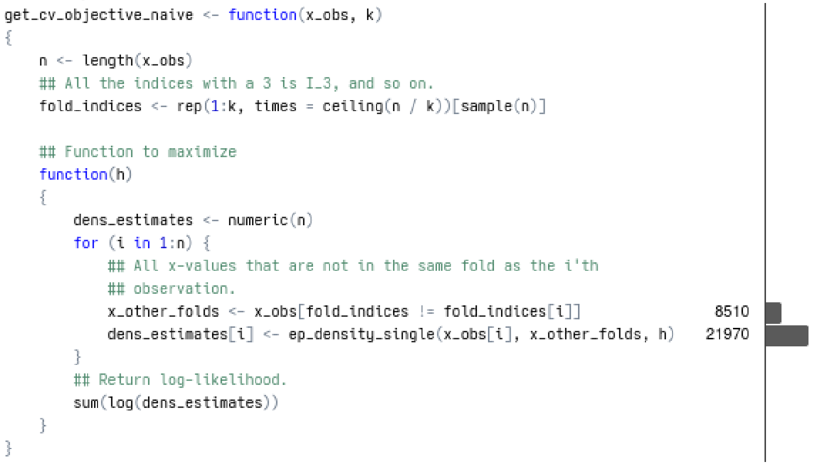
```r
get_cv_objective <- function(x_obs, k)
{
    force_all_args()
    n <- length(x_obs)
    ## All the indices with a 3 is I_3, and so on.
    fold_indices <- rep(1:k, times = ceiling(n / k))[sample(n)]

    function(h)
    {
        dens_estimates <- numeric(n)
        for (i in 1:n) {
            ## All x-values that are not in the same fold as the i'th
            ## observation (I^{-i}).
            x_other_folds <- x_obs[fold_indices != fold_indices[i]]
            dens_estimates[i] <- ep_density_single(x_obs[i],
                                                   x_other_folds,
                                                   h)
        }
        sum(log(dens_estimates))
    }
}

## Forces evaluation of all arguments
force_all_args <- function()
    as.list(parent.frame())
```

Profiling `bandwidth_cv`

10-fold cross validation on standard Gaussian data.

```r
library(profvis)
source("implementation.R", keep.source = TRUE)

n_sim <- 10000
x <- rnorm(n_sim)

profvis(bandwidth_cv(x, 10))
```

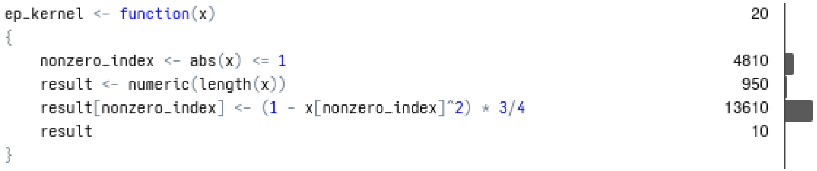
Profiling `bandwidth_cv`: Objective function



```
get_cv_objective_naive <- function(x_obs, k)
{
    n <- length(x_obs)
    ## All the indices with a 3 is I_3, and so on.
    fold_indices <- rep(1:k, times = ceiling(n / k))[sample(n)]

    ## Function to maximize
    function(h)
    {
        dens_estimates <- numeric(n)
        for (i in 1:n) {
            ## All x-values that are not in the same fold as the i'th
            ## observation.
            x_other_folds <- x_obs[fold_indices != fold_indices[i]]               8510
            dens_estimates[i] <- ep_density_single(x_obs[i], x_other_folds, h)   21970
        }
        ## Return log-likelihood.
        sum(log(dens_estimates))
    }
}
```

- Problem: We recalculate I^{-i} for each element in a given fold, and indexing is slow in R.
- Idea: Cache I^{-i} the first time we meet an element from a given fold, and reuse it for later elements.

Profiling `bandwidth_cv`: `ep_kernel`



```
ep_kernel <- function(x)                                                    20
{
    nonzero_index <- abs(x) <= 1                                          4810
    result <- numeric(length(x))                                           950
    result[nonzero_index] <- (1 - x[nonzero_index]^2) * 3/4              13610
    result                                                                  10
}
```

- Idea: Implement in C++

  (R implementation `(abs(x) <= 1) * (1 - x^2) * 0.75` is faster in some cases, but let us just go straight to C++).

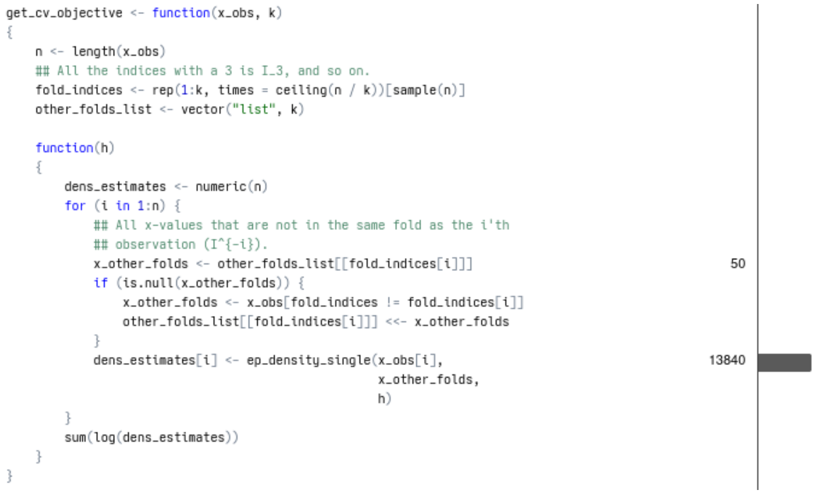
Objective function implementation with caching

```r
get_cv_objective <- function(x_obs, k)
{
    force_all_args()
    n <- length(x_obs)
    ## All the indices with a 3 is I_3, and so on.
    fold_indices <- rep(1:k, times = ceiling(n / k))[sample(n)]
    other_folds_list <- vector("list", k)
    function(h)
    {
        dens_estimates <- numeric(n)
        for (i in 1:n) {
            ## All x-values that are not in the same fold as the i'th
            ## observation (I^{-i}).
            x_other_folds <- other_folds_list[[fold_indices[i]]]
            if (is.null(x_other_folds)) {
                x_other_folds <- x_obs[fold_indices != fold_indices[i]]
                other_folds_list[[fold_indices[i]]] <<- x_other_folds
            }
            dens_estimates[i] <- ep_density_single(x_obs[i],
                                                   x_other_folds,
                                                   h)
        }
        sum(log(dens_estimates))
    }
}
```

Rcpp implementation of `ep_kernel`

```cpp
double
ep_kernel_single_cpp(double x)
{
  if (-1 < x && x < 1)
    return (1 - square(x)) * 0.75;
  else
    return 0;
}

// [[Rcpp::export]]
NumericVector
ep_kernel_cpp(const NumericVector& x)
{
  unsigned long long n = x.length();
  NumericVector y(n);
  for (unsigned long long i = 0; i < n; i++)
    y[i] = ep_kernel_single_cpp(x[i]);
  return y;
}
```

Profiling again



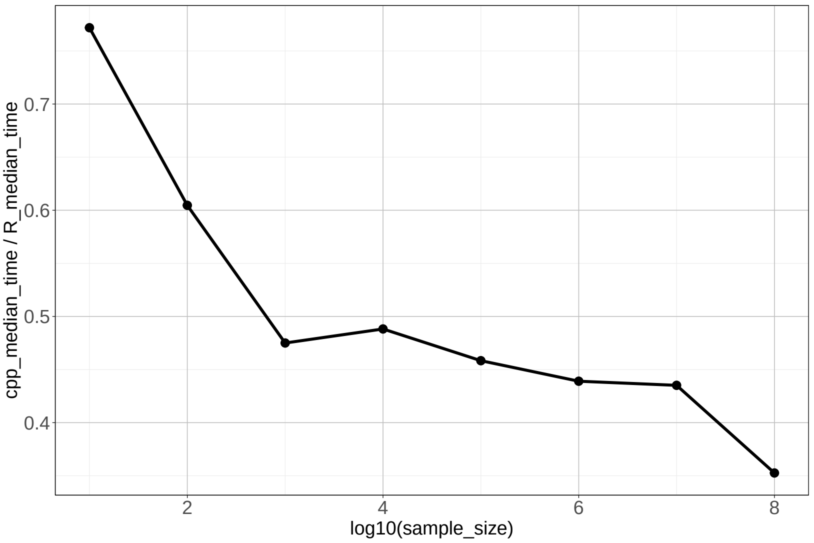
```
get_cv_objective <- function(x_obs, k)
{
    n <- length(x_obs)
    ## All the indices with a 3 is I_3, and so on.
    fold_indices <- rep(1:k, times = ceiling(n / k))[sample(n)]
    other_folds_list <- vector("list", k)

    function(h)
    {
        dens_estimates <- numeric(n)
        for (i in 1:n) {
            ## All x-values that are not in the same fold as the i'th
            ## observation (I^{-i}).
            x_other_folds <- other_folds_list[[fold_indices[i]]]                50
            if (is.null(x_other_folds)) {
                x_other_folds <- x_obs[fold_indices != fold_indices[i]]
                other_folds_list[[fold_indices[i]]] <<- x_other_folds
            }
            dens_estimates[i] <- ep_density_single(x_obs[i],                 13840
                                                   x_other_folds,
                                                   h)
        }
        sum(log(dens_estimates))
    }
}
```

Finding I^{-i} is no longer a bottleneck.

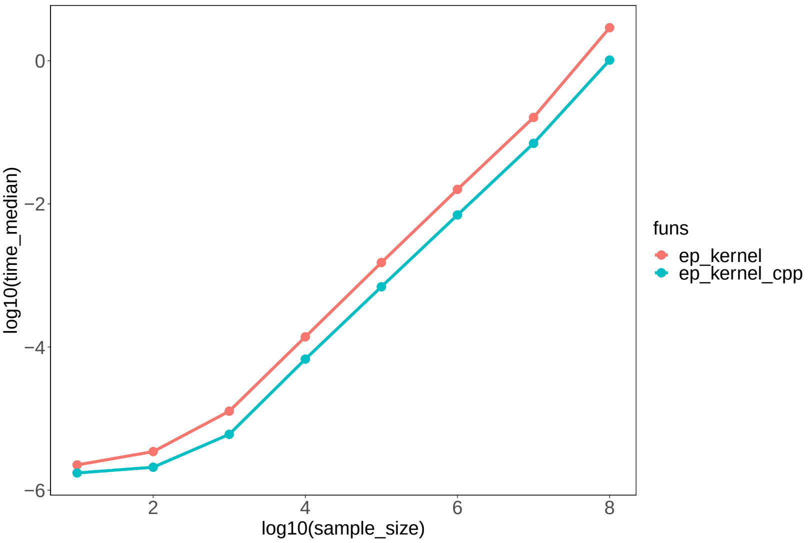
Benchmarking the `ep_kernel` implementations

```r
vectors <- lapply(10^(1:8), rnorm)
function_names <- c("ep_kernel", "ep_kernel_cpp")
call_grid <- expand.grid(funs = function_names,
                         vector_indices = 1:length(vectors))
calls <- paste0(call_grid$funs, "(vectors[[",
                call_grid$vector_indices, "]])")
expression_list <- lapply(
    calls,
    function(call_str) parse(text = call_str)[[1]]
)
benchmark <- bench::mark(exprs = expression_list,
                         min_iterations = 5,
                         check = FALSE)
```

Rcpp implementation more than halves runtime



Runtime scales linearly with sample size



CV problem with outliers

$$\begin{aligned}
\min_{j \in I^{-i}}(|x_i - x_j|) = d &\Rightarrow \hat{f}_h^{-i} = 0 \text{ for all } h \leq d \\
&\Rightarrow \log(\hat{f}_h^{-i}) = -\infty \\
&\Rightarrow \sum_{i=1}^{n} \log(\hat{f}_h^{-i}) = -\infty \\
&\Rightarrow \hat{h}_{CV(k)} = \mathrm{argmax}_h \sum_{i=1}^{n} \log(\hat{f}_h^{-i}) > d.
\end{aligned}$$

Hence

$$\hat{h}_{CV(k)} > \max_{i=1,\dots,n} \min_{j \in I^{-i}}(|x_i - x_j|).$$

One x_i far from all points in the other folds I^{-i} leads to a large $\hat{h}_{CV(k)}$ (oversmoothing).

Other approach: Minimize AMISE

- The Asymptotic Mean Integrated Squared Error (that is, the asymptotically dominating terms of the integral of the MSE) is minimized by

$$h_n = \left(\frac{\|K\|_2^2}{\|f_0''\|_2^2 \sigma_K^4} \right)^{1/5} n^{-1/5}.$$

- h_n is thus a reasonable choice of bandwidth for kernel density estimation... if we knew $\|f_0''\|_2^2$.

Plug-in estimate

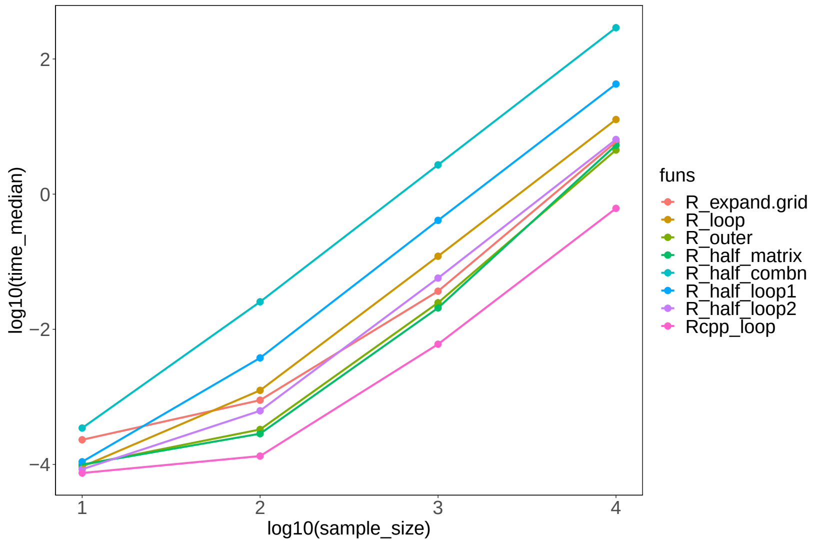
- Use Silverman's rule of thumb to choose initial bandwidth $r = 0.9\tilde{\sigma} n^{-1/5}$, where $\tilde{\sigma} = \min\{\hat{\sigma}, \mathrm{IQR}/1.34\}$.
- Estimate $\|f_0''\|_2^2$ using a Gaussian kernel H and bandwidth r:

$$\begin{aligned}\|\tilde{f}''\|_2^2 &= \frac{1}{n^2 r^6} \sum_{i=1}^{n} \sum_{j=1}^{n} \int H''\left(\frac{x - x_i}{r}\right) H''\left(\frac{x - x_j}{r}\right) dx \\ &= \frac{1}{n^2(\sqrt{2}r)^5} \sum_{i=1}^{n} \sum_{j=1}^{n} \phi^{(4)}\left(\frac{X_i - X_j}{\sqrt{2}r}\right)\end{aligned}$$

  (Bandwidth Selection: Classical or Plug-In?, Clive R. Loader, The Annals of Statistics, Apr., 1999, Vol. 27, No. 2, pp. 415-438)

- Use $\|\tilde{f}''\|_2^2$ to calculate $\hat{h}_n$.

Long story short: Rcpp wins



Final solution in Rcpp (highest level)

```cpp
double
d2f0_two_norm_squared_estimate_cpp(const NumericVector& x, double r)
{
  unsigned long long n = x.length();
  double result = 0;
  double rsqrt2 = r * sqrt(2);
  for (unsigned long long i = 0; i < n - 1; i++)
    for (unsigned long long j = i + 1; i < j && j < n; j++)
      result += d4_std_gaussian_density_single_cpp((x[j] - x[i]) / rsqrt2);
  return (2 * result + n * d4_std_gaussian_density_0) /
    (square(n) * pow(rsqrt2, 5));
}
```

Final solution in Rcpp (density)

```cpp
double
d4_std_gaussian_density_single_cpp(double x)
{
  double x_sq = square(x);
  return exp(-x_sq/2) * ((x_sq - 6) * x_sq + 3) / sqrt2pi;
}

NumericVector
d4_std_gaussian_density_cpp(const NumericVector& x)
{
  unsigned long long n = x.length();
  NumericVector y(n);
  for (unsigned long long i = 0; i < n; i++)
    y[i] = d4_std_gaussian_density_single_cpp(x[i]);
  return y;
}
```

Highest level of R implementation

```r
bandwidth_plug_in <- function(x_obs)
    (15 / (d2f0_two_norm_squared_estimate(x_obs) * length(x_obs)))^0.2
```

- We must implement `d2f0_two_norm_squared_estimate`.
- In R:

```r
## Estimate of the squared 2-norm of f0''.
d2f0_two_norm_squared_estimate <- function(x_obs,
                                           r = bandwidth_silverman(x_obs),
                                           diff_vec = diff_vec_expand.grid)
    sum(d4_std_gaussian_density(diff_vec(x_obs) / (sqrt2 * r))) /
        (length(x_obs)^2 * (sqrt2 * r)^5)
```

where `diff_vec` must return the vector of all pairwise differences of elements in `x_obs` (due to symmetry of the Epanechnikov kernel, we can actually get away with only calculating half of the differences).

Many ways to implement `diff_vec`...

```r
## Returns vector of all differences of pairs.
diff_vec_expand.grid <- function(x_obs)
{
    x_grid <- expand.grid(Xi = x_obs, Xj = x_obs)
    x_grid[, 1] - x_grid[, 2]
}

## Same as above, but uses for loop
diff_vec_loop <- function(x_obs)
{
    n <- length(x_obs)
    res <- numeric(n^2)
    count <- 1
    for (i in 1:n) {
        for (j in 1:n) {
            res[count] <- x_obs[i] - x_obs[j]
            count <- count + 1
        }
    }
    res
}

## Same as above, but uses "outer" instead of "expand.grid" and returns matrix.
diff_matrix_outer <- function(x_obs)
    outer(x_obs, x_obs, FUN = "-")

## Returns vector version of matrix result from above.
diff_vec_outer <- function(x_obs)
    as.vector(diff_matrix_outer(x_obs))
```

. . . and a couple more.

```r
## One half of differences, since epa-kernel is symmetric
diff_half_combn <- function(x_obs)
    combn(x_obs, 2, FUN = diff)

diff_half_outer <- function(x_obs)
{
    diff_mat <- diff_matrix_outer(x_obs)
    diff_mat[upper.tri(diff_mat)]
}
```

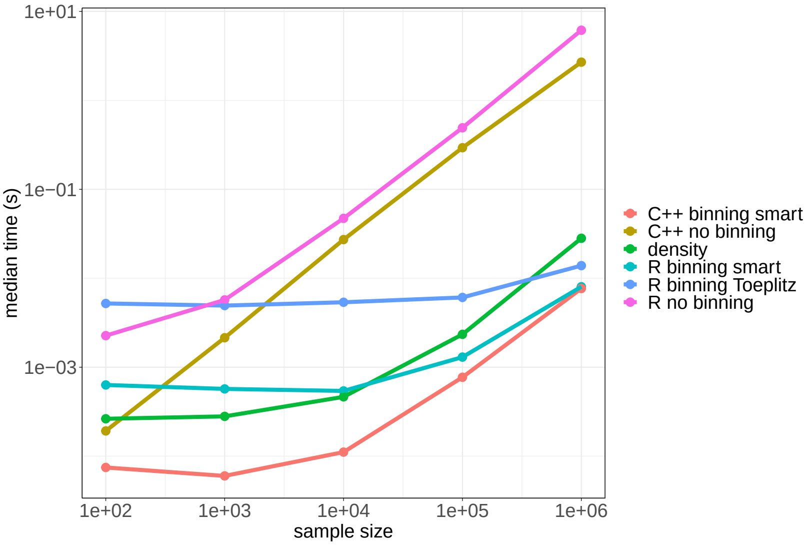
What else could be done?

- Implementation for general kernel (e.g., allow user to supply kernel S3 object, and the program has fall-back option of numerical integration if $\|\tilde{f}''\|_2^2$ is not implemented by the user).

Source on binning

Sections 5.2.1 and 5.3.2 of *Nonparametric Kernel Density Estimation and Its Computational Aspects* by Artur Gramacki (Springer 2018))

Old benchmark (without `range_cpp`)


median time (s)
1e+01
1e−01
1e−03
sample size
1e+02
1e+03
1e+04
1e+05
1e+06
C++ binning smart
C++ no binning
density
R binning smart
R binning Toeplitz
R no binning