Kernel Density Estimation

Assignment 1

Lucas Støjko Andersen

University of Copenhagen

November 8, 2022

Kernel Density Estimation

Kernels

Let $K : \mathbb{R} \longrightarrow [0, \infty)$ with properties

$$K(x) = K(-x), \quad \forall x \in \mathbb{R} \tag{1}$$

$$1 = \int_{\mathbb{R}} K(x) dx \tag{2}$$

and $X_1, X_2, \dots X_n$ be random variables with density f. Then

$$\hat{f}(x) = \frac{1}{n\lambda} \sum_{i=1}^{n} K\left(\frac{x - x_i}{\lambda}\right) \tag{3}$$

is the kernel density estimate of f with bandwidth $\lambda > 0$ and kernel K.

Implementation of Kernel Density Estimate

```r
R_dens_for <- function(x, p, kernel, bandwidth) {
  m <- length(p)
  n <- length(x)
  result <- numeric(m)
  for(i in 1:m) {
    for(j in 1:n) {
      result[i] <- result[i] + kernel((p[i] - x[j]) / bandwidth)
    }
  }
  result / (n * bandwidth)
}
```

Testing the Implementation

Epanechnikov Kernel

Use the kernel density estimate on the *epanechnikov* kernel given by

$$K(x) = \frac{3}{4}(1 - x^2)1_{[0,1]}(x). \tag{4}$$

Implemented in `R` as

```
e_kernel <- function(x) {
  0.75 * (1 - x^2) * (abs(x) <= 1)
}
```

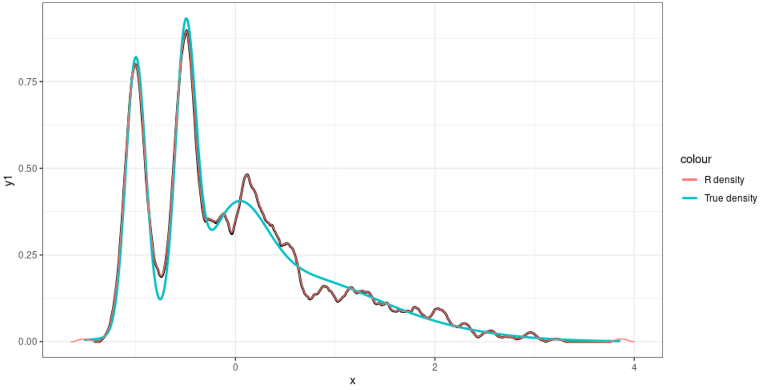
The Test Case

Gaussian Mixture

```
x <- c(rnorm(200, -1, 0.1),
       rnorm(200, -0.5, 0.1),
       rnorm(200, 0, 0.3),
       rnorm(200, 0.5, 0.7),
       rnorm(200, 1, 1))

funky_density <- function(x) {
  dnorm(x, -1, 0.1) / 5 +
    dnorm(x, -0.5, 0.1) / 5 +
    dnorm(x, 0, 0.3) / 5 +
    dnorm(x, 0.5, 0.7) / 5 +
    dnorm(x, 1, 1) / 5
}
```

Plotting the Densities

With $\lambda = 0.1$ we plot the estimated density together with `R`'s default `density` implementation and the true density.



Bandwidth Selection

Leave-One-Out Cross Validation

We persue bandwidth selection through leave-one-out cross validation. Let

$$\hat{f}_\lambda^{-i} = \frac{1}{(n-1)\lambda} \sum_{j \neq i}^{n} K\left(\frac{x_i - x_j}{\lambda}\right). \tag{5}$$

We then select the bandwidth

$$\hat{\lambda}_{\mathrm{CV}} = \arg\max_{\lambda > 0} \sum_{i=1}^{n} \log \hat{f}_\lambda^{-i}. \tag{6}$$

Implementation of LOOCV Bandwidth Selection

Bandwidth Selection and Oracle Bandwidth as Comparison

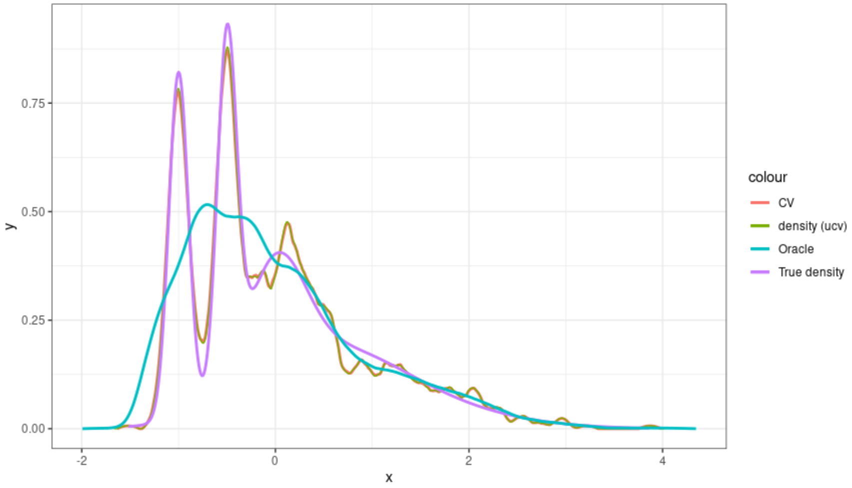
```r
bw_cv_R2 <- function(x, kernel, max_bw = 2) {
  cv_func <- function(l) {
    if(l < .Machine$double.eps) Inf
    n <- length(x)
    K <- numeric(n)
    for(i in 1:n) {
      K[i] <- sum(kernel((x[i] - x[-i]) / l))
    }
    cv <- sum(log(K[K > .Machine$double.eps]))
    n * log((n - 1) * l) - cv
  }
  suppressWarnings(optimize(cv_func, c(0, max_bw)))$minimum
}

bw_oracle <- function(x, kernel) {
  n <- length(x)
  K <- integrate(function(x) kernel(x)^2, -Inf, Inf)$value
  sigma2 <- integrate(function(x) kernel(x) * x^2, -Inf, Inf)$value
  sigma <- min(sd(x), IQR(x) / 1.34)
  (8 * sqrt(pi) * K / (3 * sigma2^2))^(1/5) * sigma * n^(-1/5)
}
```

Implementation of Kernel Density Estimate with Bandwidth Selection

```r
dens_R1 <- function(x,
                    kernel,
                    bandwidth,
                    ...,
                    points = 512L) {
  if(is.function(bandwidth)) {
    bw <- bandwidth(x, kernel, ...)
  } else {
    bw <- bandwidth
  }
  p <- seq(min(x) - bw, max(x) + bw, length.out = points)
  y <- R_dens_for(x, p, kernel, bw)
  structure(
    list(
      x = p,
      y = y,
      bw = bw
    ),
    class = "dens"
  )
}
```

Plotting the Density Estimate with Bandwidth Selection

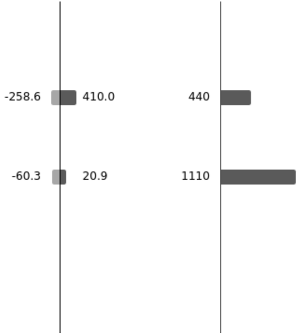
The LOOCV bandwidth is 0.1155167 and the oracle bandwidth is 0.4746436.



Profiling the Implementation

Profiling the Kernel Density Estimation

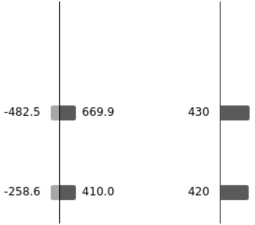
```r
dens_R1 <- function(x,
                    kernel,
                    bandwidth,
                    points = 512L) {
  if(is.function(bandwidth)) {
    bw <- bandwidth(x, kernel)
  } else {
    bw <- bandwidth
  }
  p <- seq(min(x) - bw, max(x) + bw, length.out = points)
  y <- R_dens_for(x, p, kernel, bw)
  structure(
    list(
      x = p,
      y = y,
      bw = bw
    ),
    class = "dens"
  )
}
```



Profiling the Implementation

Profiling the Bandwidth Selection

```r
bw_cv_R2 <- function(x, kernel) {
  cv_func <- function(l) {
    if(l < .Machine$double.eps) Inf
    n <- length(x)
    K <- numeric(n)
    for(i in 1:n) {
      K[i] <- sum(kernel((x[i] - x[-i]) / l))
    }
    cv <- sum(log(K[K > .Machine$double.eps]))
    n * log((n - 1) * l) - cv
  }
  suppressWarnings(optimize(cv_func, c(0, 2)))$minimum
}
```

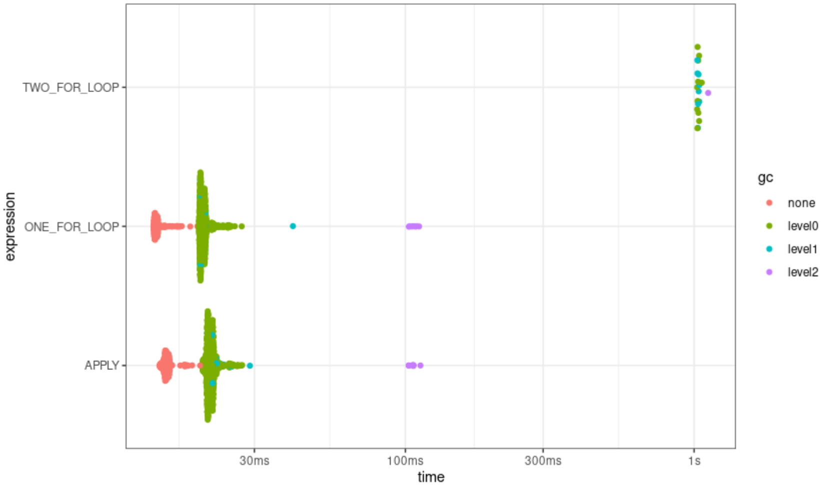


Alternative Implementations of Calculating the Density Estimate

```
R_dens <- function(x, p, kernel, bandwidth) {
  m <- length(p)
  n <- length(x)
  result <- numeric(m)
  for(i in 1:m) {
    result[i] <- sum(kernel((p[i] - x) / bandwidth))
  }
  result / (n * bandwidth)
}

R_dens_for <- function(x, p, kernel, bandwidth) {
  m <- length(p)
  n <- length(x)
  result <- numeric(m)
  for(i in 1:m) {
    for(j in 1:n) {
      result[i] <- result[i] + kernel((p[i] - x[j]) / bandwidth)
    }
  }
  result / (n * bandwidth)
}

R_dens_apply <- function(x, p, kernel, bandwidth) {
  app_func <- function(t) sum(kernel((t - x) / bandwidth))
  result <- sapply(p, app_func)
  result / (length(x) * bandwidth)
}
```

Benchmarking Alternative Implementations using `bench` Package

Plot



Benchmarking Alternative Implementations using bench Package

Table

	expression	median	mem_alloc
	<bch:expr>	<bch:tm>	<bch:byt>
1	ONE_FOR_LOOP	19.61ms	35.5MB
2	APPLY	20.83ms	35.4MB
3	TWO_FOR_LOOP	1.03s	79.1KB

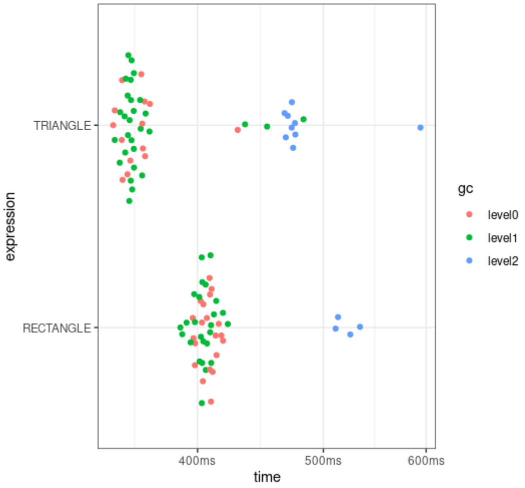
Alternative LOOCV Implementations

Let $K_{i,j} = K((x_i - x_j)/\lambda)$ and notice $K_{i,j} = K_{j,i}$.

```
bw_cv_R <- function(x, kernel, max_bw = 2) {
  cv_func <- function(l) {
    if(l < .Machine$double.eps) Inf
    n <- length(x)
    K <- numeric(n)
    for(i in 2:n) {
      index <- 1:(i - 1)
      tmp <- kernel((x[i] - x[index]) / l)
      K[i] <- K[i] + sum(tmp)
      K[index] <- K[index] + tmp[index]
    }
    cv <- sum(log(K[K > .Machine$double.eps]))
    n * log((n - 1) * l) - cv
  }
  suppressWarnings(optimize(cv_func, c(0, max_bw)))$minimum
}
```

Alternative LOOCV Implementation Benchmarks

Plot



Alternative LOOCV Implementation Benchmarks

Table

	expression	median	mem_alloc
	<bch:expr>	<bch:tm>	<bch:byt>
1	TRIANGLE	357ms	433MB
2	RECTANGLE	407ms	738MB

Improving Further with RCPP Package

Implementing the kernel in RCPP

Implementing the kernel in RCPP could improve performance per the profiling.

```
// [[Rcpp::export]]
NumericVector e_kernel_cpp(NumericVector x) {
  int n = x.size();
  NumericVector result(n);
  for(int i = 0; i < n; ++i)
    result[i] = std::abs(x[i]) <= 1 ? 0.75 * (1 - x[i] * x[i]) : 0;
  return result;
}
```

Improving Further with RCPP Package

Calling Kernel from R in RCPP

```
// [[Rcpp::export]]
NumericVector dens_rcpp_partial(NumericVector x,
                                Function kernel,
                                double bandwidth,
                                NumericVector points) {
  int n = x.size(), m = points.size();
  NumericVector result(m);
  for(int i = 0; i < m; ++i) {
    NumericVector call = kernel((points[i] - x) / bandwidth);
    result[i] = sum(call) / (n * bandwidth);
  }
  return result;
}
```

Improving Further with RCPP Package

Calling Kernel from R in RCPP

```
// [[Rcpp::export]]
double bw_cv_rcpp_partial(NumericVector x,
                          Function kernel,
                          double bandwidth) {
  int n = x.size();
  NumericVector K(n);
  double result;
  for(int i = 1; i < n; ++i) {
    Range r = Range(0, i - 1);
    NumericVector tmp = kernel((x[r] - x[i]) / bandwidth);
    for(int j = 0; j < i; ++j)
      K[j] += tmp[j], K[i] += tmp[j];

  }
  for(int s = 0; s < n; ++s)
    if(K[s] > std::numeric_limits<double>::min()) result += std::log(K[s]);
  return n * log((n - 1) * bandwidth) - result;
}
```

Improving Further with RCPP Package

Full RCPP Implementation

```
// [[Rcpp::export]]
NumericVector dens_rcpp(NumericVector x,
                        double bandwidth,
                        NumericVector points) {
  int n = x.size(), m = points.size();
  NumericVector result(m);
  for(int i = 0; i < m; ++i) {
    NumericVector call = e_kernel_cpp((points[i] - x) / bandwidth);
    result[i] = sum(call) / (n * bandwidth);
  }
  return result;
}
```

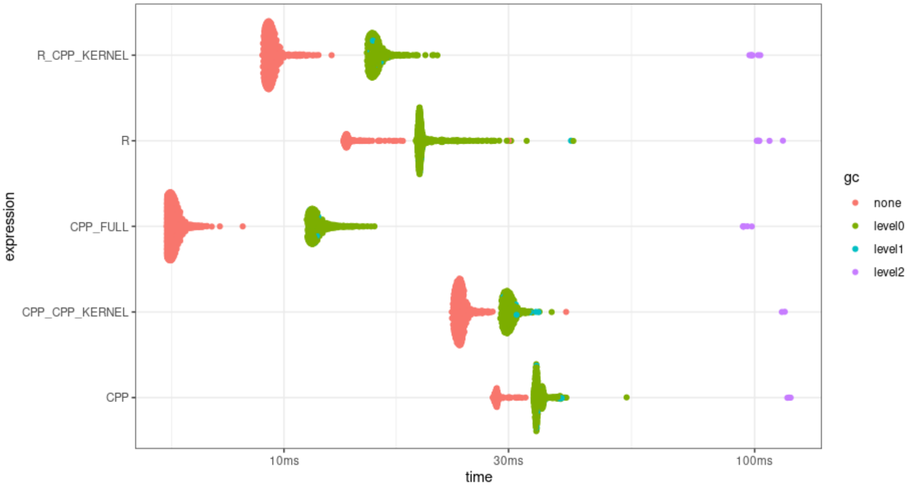
Improving Further with RCPP Package

Full RCPP Implementation

```cpp
// [[Rcpp::export]]
double bw_cv_rcpp(NumericVector x,
                  double bandwidth) {
  int n = x.size();
  NumericVector K(n);
  double result;
  for(int i = 1; i < n; ++i) {
    Range r = Range(0, i - 1);
    NumericVector tmp = e_kernel_cpp((x[i] - x[r]) / bandwidth);
    for(int j = 0; j < i; ++j)
      K[j] += tmp[j], K[i] += tmp[j];
  }
  for(int s = 0; s < n; ++s)
    if(K[s] > std::numeric_limits<double>::min()) result += std::log(K[s]);
  return n * std::log((n - 1) * bandwidth) - result;
}
```

Benchmarking RCPP Density Calculation

Plot



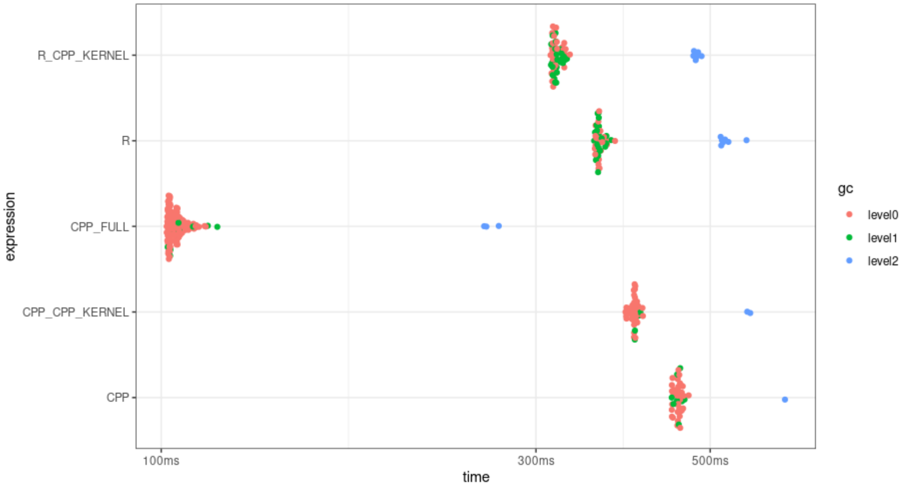
Benchmarking RCPP Density Calculation

Table

	expression	median	mem_alloc
	<bch:expr>	<bch:tm>	<bch:byt>
1	CPP_FULL	5.73ms	15.7MB
2	R_CPP_KERNEL	9.28ms	18.2MB
3	R	13.62ms	35.5MB
4	CPP_CPP_KERNEL	23.66ms	18.2MB
5	CPP	28.32ms	35.4MB

Benchmarking RCPP Bandwidth Selection

Plot



Benchmarking RCPP Bandwidth Selection

Table

	expression	median	mem_alloc
	<bch:expr>	<bch:tm>	<bch:byt>
1	CPP_FULL	5.73ms	15.7MB
2	R_CPP_KERNEL	9.28ms	18.2MB
3	R	13.62ms	35.5MB
4	CPP_CPP_KERNEL	23.66ms	18.2MB
5	CPP	28.32ms	35.4MB

Improving Further Using R's C API

Implementation of Density Calculation

```c
#include <Rinternals.h>
#include <R.h>

SEXP C_dens(SEXP x, SEXP p, SEXP kernel, SEXP bw, SEXP rho) {
  int n = length(x), m = length(p);
  SEXP dens = PROTECT(allocVector(REALSXP, m));
  SEXP tmp = PROTECT(allocVector(REALSXP, n));
  SEXP K_Call = PROTECT(lang2(kernel, R_NilValue));
  double *x_ = REAL(x), *p_ = REAL(p), *tmp_ = REAL(tmp), *dens_ = REAL(dens);
  double bw_ = REAL(bw)[0];

  memset(dens_, 0, sizeof(double) * m);

  for(int i = 0; i < m; ++i) {
    for(int j = 0; j < n; ++j)
      tmp_[j] = (p_[i] - x_[j]) / bw_;
    SETCADR(K_Call, tmp);
    SEXP result = eval(K_Call, rho);
    double *result_ = REAL(result);
    for(int j = 0; j < n; ++j)
      dens_[i] += result_[j];
    dens_[i] /= n * bw_;
  }
  UNPROTECT(3);
  return dens;
}
```

Improving Further Using R's C API

Implementation of LOOCV

```c
#include <Rinternals.h>
#include <R.h>
#include <float.h>

SEXP C_cv(SEXP x, SEXP fn, SEXP lambda, SEXP rho) {
  if(REAL(lambda)[0] < DBL_EPSILON) return ScalarReal(INFINITY);
  int n = length(x);
  SEXP K = PROTECT(allocVector(REALSXP, n));
  SEXP fn_call = PROTECT(lang2(fn, R_NilValue));
  SEXP out = PROTECT(ScalarReal(0.0));
  double *x_ = REAL(x), *K_ = REAL(K), *out_ = REAL(out);
  double h_ = REAL(lambda)[0];
  memset(K_, 0, sizeof(double) * n);
  for(int i = 1; i < n; ++i) {
    SEXP tmp = PROTECT(allocVector(REALSXP, i));
    double *tmp_ = REAL(tmp);
    for(int k = 0; k < i; ++k)
      tmp_[k] = (x_[i] - x_[k]) / h_;
    SETCADR(fn_call, tmp);
    SEXP s = eval(fn_call, rho);
    double *s_ = REAL(s);
    UNPROTECT(1);
    for(int j = 0; j < i; ++j) {
      K_[i] += s_[j];
      K_[j] += s_[j];
    }
  }
  for(int i = 0; i < n; ++i)
    if(K_[i] > DBL_EPSILON) *out_ += log(K_[i]);
  *out_ = n * log((n - 1) * h_) - (*out_);
  UNPROTECT(3);
  return out;
}
```

Improving Further Using R's C API

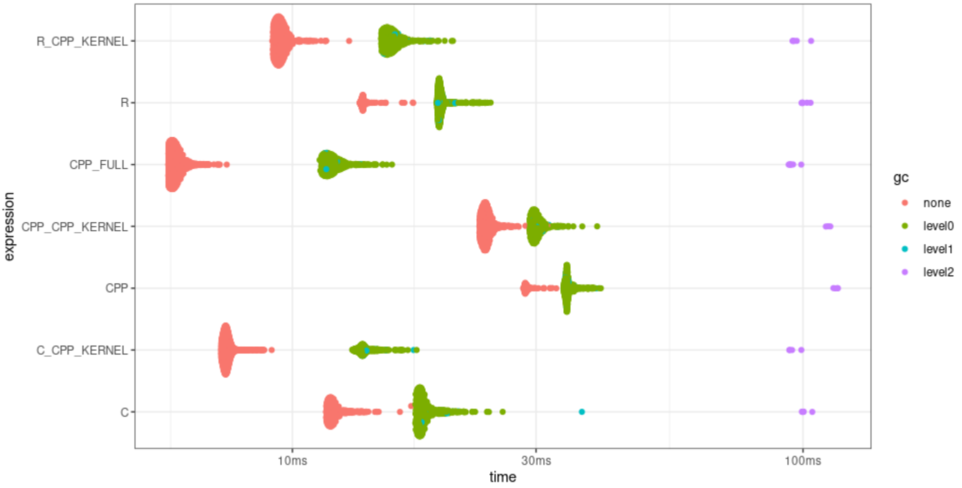
Calling the C Code

Compile to shared library using `R CMD SHLIB` command. If used in a package a little more work is required. For now just used `dyn.load` function in R to link the shared library.

```
bw_cv <- function(x, kernel, max_bw = 2) {
  cv_func <- function(l) .Call("C_cv", x, kernel, l, environment())
  suppressWarnings(optimize(cv_func, c(0, max_bw)))$minimum
}
```

Benchmarking All Density Calculation Implementations

Plot



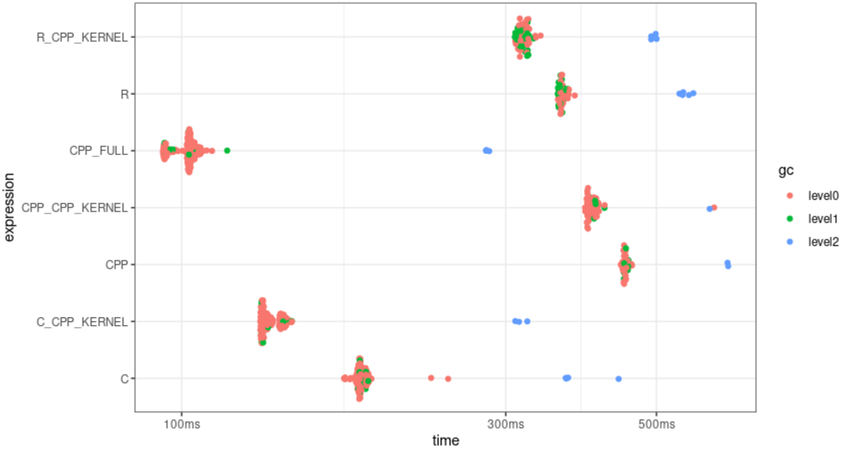
Benchmarking All Density Calculation Implementations

Table

	expression	median	mem_alloc
	<bch:expr>	<bch:tm>	<bch:byt>
1	CPP_FULL	5.81ms	15.7MB
2	C_CPP_KERNEL	7.42ms	10.4MB
3	R_CPP_KERNEL	9.39ms	18.2MB
4	C	11.89ms	27.6MB
5	R	13.74ms	35.5MB
6	CPP_CPP_KERNEL	23.9ms	18.2MB
7	CPP	28.7ms	35.4MB

Benchmarking All LOOCV Implementations

Plot



Benchmarking All LOOCV Implementations

Table

	expression	median	mem_alloc
	<bch:expr>	<bch:tm>	<bch:byt>
1	CPP_FULL	103ms	124MB
2	C_CPP_KERNEL	133ms	163MB
3	C	183ms	279MB
4	R_CPP_KERNEL	320ms	317MB
5	R	363ms	433MB
6	CPP_CPP_KERNEL	401ms	162MB
7	CPP	450ms	278MB

Final implementation

```
dens <- function(x,
                 kernel,
                 bandwidth = bw_cv,
                 ...,
                 points = 512L) {
  if(is.function(bandwidth)) {
    bw <- bandwidth(x, kernel, ...)
  } else {
    bw <- bandwidth
  }
  p <- seq(min(x) - bw, max(x) + bw, length.out = points)
  y <- .Call("C_dens", x, p, kernel, bw, environment())
  structure(
    list(
      x = p,
      y = y,
      bw = bw
    ),
    class = "dens"
  )
}
```