Journal of Statistical Software

January 2022, Volume 101, Issue 3. *doi: 10.18637/jss.v101.i03*

Fast Kernel Smoothing in R with Applications to Projection Pursuit

David P. Hofmeyr
Stellenbosch University

Abstract

This paper introduces the R package **FKSUM**, which offers fast and exact evaluation of univariate kernel smoothers. The main kernel computations are implemented in C++, and are wrapped in simple, intuitive and versatile R functions. The fast kernel computations are based on recursive expressions involving the order statistics, which allows for exact evaluation of kernel smoothers at all sample points in log-linear time. In addition to general purpose kernel smoothing functions, the package offers purpose built and ready-to-use implementations of popular kernel-type estimators. On top of these basic smoothing problems, this paper focuses on projection pursuit problems in which the projection index is based on kernel-type estimators of functionals of the projected density.

Keywords: kernel smoothing, nonparametric, density estimation, regression, projection pursuit, independent component analysis, R.

1. Introduction

Kernels offer an extremely flexible way of estimating (usually) smooth functions nonparametrically. At the essence of kernel smoothing, and indeed many nonparametric methods, is the simple concept of a local average around a point, x; that is, a weighted average of some observable quantities, those of which closest to x being given the highest weights. Suppose our objective is the estimation of some structure (e.g., a function), which we cannot measure directly, and we instead make (indirect) observations subject to random error. If we can assume that these observations offer (close to) unbiased measurements of the function of interest, then in an ideal sampling scenario we would be able to make multiple such *noisy* measurements at each point of interest, so that highly efficient estimation can be achieved by taking the averages of these. In practice, however, we seldom have such control over how we sample observations, or there may be cost constraints which severely limit such direct approaches. Instead, we rely on the very simple observation that if the function of interest is continuous, then it will not change too substantially over a small region. Therefore, all observations which are in a neighborhood of a point of interest should also provide reasonable information about the target. Averaging our observations, but placing almost all weight on those points in a neighborhood of the target, is therefore very well motivated.

Kernels provide an intuitive means for achieving such local averaging. In the most simple context (although in some contexts even the following may be relaxed), a kernel is simply a non-negative function which vanishes quickly as the magnitude of its argument increases. Kernels can be used as weighting functions if applied to the pairwise distances between points, so that this vanishing tendency ensures that weights associated with large distances between points are very small. Any weighted average arising from kernel weights will therefore only apply large weights to those points which are relatively near to the point of interest, i.e., in its neighborhood. Furthermore, a simple rescaling of the function's domain, with a so-called *bandwidth*, allows one to control how quickly this vanishing occurs, and hence how large is this neighborhood.

At the essence of kernel methods in statistics, then, lies the evaluation of sums of the form

$$S(x \mid \mathbf{x}, \boldsymbol{\omega}) := \sum_{j=1}^{n} K\left(\frac{x_j - x}{h}\right)\omega_j, \tag{1}$$

where $K(\cdot)$ is the kernel function, x is a point of interest, $\mathbf{x} = (x_1, \dots, x_n)$ is a vector of observed sample points and $\boldsymbol{\omega} = (\omega_1, \dots, \omega_n)$ is for now an arbitrary vector of coefficients associated with the observations. These could be the noisy measurements of the structure/function being estimated, but as we will encounter in the remainder, may represent a variety of options. The parameter h is the bandwidth, and it should be clear that if h is relatively small then it will increase the magnitude of the arguments in $K(\cdot)$, so that the relative sizes of the associated weights will more heavily emphasize points close to x. The bandwidth is in a more general context referred to as a smoothing parameter, in that large values of h lead to closer to uniform weights, and hence the total sum, $S(x \mid \mathbf{x}, \boldsymbol{\omega})$, will vary less for different points, x (i.e., will represent a smoother function of x). Arguably the most important areas of research in the context of kernel-type estimation is in the appropriate selection of the bandwidth, h, and in the design of efficient algorithms for evaluating $S(\cdot \mid \mathbf{x}, \boldsymbol{\omega})$ for a large collection of evaluation points. In fact, as we will encounter, it is frequently necessary to compute these sums for all of the observations, $\mathbf{x}$, themselves. Naïve evaluation of all such sums has computational complexity which is quadratic in n, which is prohibitive for even moderate sized problems. Some kernels with bounded support (those which take the value zero outside some compact set), as well as the Laplace kernel, allow so-called *fast sum updating* (Langrené and Warin 2019; Fan and Marron 1994; Chen 2006), which means that these sums can be computed recursively, leading to log-linear computational complexity (the log factor arising from the fact that the observations and evaluation points must be sorted). A similar recursive approach was recently discussed for the class of kernels given by the product of a polynomial and the Laplace kernel (Hofmeyr 2021). Applications of the fast sum updating approach to products was also mentioned by Langrené and Warin (2019), although details are not given. The bounded support kernels tend to enjoy high *efficiency*, which relates to them inducing relatively low asymptotic mean integrated squared error (AMISE) when used for estimation. A limitation of these kernels is that they can result in all weights at a point being exactly zero. This may cause problems when using the estimated functions for prediction on some test data which include points outside of the range of the observations. They can also lead to less stable cross-validation for objectives such as maximum pseudo-likelihood, for the same reason. Other methods which are used for moderate-to-large sized problems rely on approximations, with popular examples being the fast Gauss and Fourier transforms (FGT, FFT, Yang, Duraiswami, Gumerov, and Davis 2003; Silverman 1982), and binning (Scott and Sheather 1985; Hall and Wand 1996).

In this paper we will discuss the R (R Core Team 2021) package **FKSUM** (Hofmeyr 2022), which is available from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=FKSUM`. The package provides an implementation of the method described by Hofmeyr (2021) for fast and exact computation of sums as in Equation 1. The main kernel computations are implemented in C++ (Stroustrup 2013), and accessed through R using the **Rcpp** package (Eddelbuettel and François 2011). Because of their popularity in nonparametric statistics, there are numerous packages which provide implementations of simple kernel smoothing methods. Popular examples include **KernSmooth** (Wand 2021), **ks** (Duong 2021), and **sm** (Bowman and Azzalini 2021), all of which are available through CRAN. In addition R's base **stats** package (R Core Team 2021) includes basic kernel smoothing functions. As far as we are aware, however, no existing R packages offer exact evaluation of kernel smoothers at all sample points in faster than quadratic running time. In addition to the basic univariate kernel estimators, **FKSUM** offers implementations of multiple projection pursuit methods, including independent component analysis (ICA, Hyvärinen and Oja 2000); projection pursuit regression (PPR, Friedman and Stuetzle 1981); and minimum density hyperplane estimation for clustering (MDPP, Pavlidis, Hofmeyr, and Tasoulis 2016). Multiple R packages offer a variety of ICA models, including **fastICA** (Marchini, Heaton, and Ripley 2021), **PearsonICA** (Karvanen and Koivunen 2002), **ProDenICA** (Hastie and Tibshirani 2010) and **JADE** (Miettinen, Nordhausen, and Taskinen 2017). Arguably the most principled ICA objective is to minimize the mutual information in the estimated components, which is equivalent to minimizing their individual differential entropies. Of the existing R implementations of which we are aware, **ProDenICA** is the only one which optimizes a direct estimate of this objective. **ProDenICA** achieves computational viability of the objective via a binning-type approach to speed up the necessary nonparametric estimation task. The fast kernel smoothing in **FKSUM** means no such approximations are needed in order to estimate and optimize entropy. The standard PPR model is implemented in R's base **stats** package, while the **gsg** package (Morrissey and Sakrejda 2014) provides generalizations to binary and Poisson responses. The implementation in **FKSUM** provides functionality for the use of an arbitrary differentiable loss function, and so also offers considerable customization. In addition, because the optimization is performed using gradient descent instead of iterative re-weighted least squares, the implementation in **FKSUM** has a computational advantage in high dimensional applications over existing implementations. Finally, MDPP is implemented, along with other cluster motivated projection pursuit models, in the package **PPCI** (Hofmeyr and Pavlidis 2019). The only other packages to combine projection pursuit with clustering explicitly, as far as we are aware, are **ProjectionBasedClustering** (Thrun and Ultsch 2020) and **Pursuit** (Ossani and Cirillo 2021). The latter of these includes a range of projection pursuit models, including multiple exploratory methods, which may be seen as alternatives to ICA in certain applications.

This paper has two main objectives. The first is to provide the reader with a basic understanding of the methods implemented in the package, as well as the know-how for their use. The second is to provide more advanced readers with the tools to implement their own methods, or existing methods based on kernel smoothing which lie beyond the scope of the package. We discuss such implementations on two scales. We cover a very simple example, kernel regression; as well as an in-depth description of the implementation of projection pursuit regression using the general purpose functions provided in the package. This example should give the more advanced reader coverage of the majority of challenges which they are likely to encounter in implementing their own projection pursuit methods, or methods outside the scope of the package.

1.1. Getting started

The **FKSUM** package can be installed and loaded from within the R console with the commands

```
R> install.packages("FKSUM")
R> library("FKSUM")
```

A brief introduction to the basic functionality of the package may then be accessed with the command

```
R> help(FKSUM)
```

The purpose built implementations offered in the package are kernel density estimation (`fk_density`), kernel regression (`fk_regression`), independent component analysis (`fk_ICA`), projection pursuit regression (`fk_ppr`) and minimum density hyperplanes (`fk_mdh`). Details for the use of any function can be obtained from within the R console with the command `help(<function name>)`, e.g., `help(fk_ICA)`.

The remainder of the paper is organized as follows. In Section 2 we introduce the use of the general purpose function for performing efficient and exact kernel smoothing. We go on to provide a simple but instructive example of its use, in terms of a simple kernel regression estimator. In Section 3 we give a thorough introduction to the general projection pursuit problem, before discussing the models and implementations in the package. In Section 3.5 we provide a detailed discussion, with reference to the implementation of projection pursuit regression, which should provide the reader with sufficient know-how for implementing their own projection pursuit methods which require efficient kernel evaluations. Finally, we give some concluding remarks in Section 4.

It should be noted that all outputs presented in Section 3 were obtained by running the associated code in R version 4.1.2 on a `x86_64-w64-mingw32` platform in Windows 10. Despite carefully setting random seeds, we have observed minor discrepancies when running the code in different versions of R, and on different platforms. However, the general performance and the relative comparisons with other implementations, where relevant, should be similar regardless of computing environment.

2. Fast kernel computations with FKSUM

In this section we introduce the general approach to perform kernel smoothing using the **FKSUM** package. Illustrations using the general purpose function `fk_sum()`, which provides exact evaluation of sums of the form in Equation 1, will be provided. The function runs in $\mathcal{O}(n\log(n) + m\log(m))$ time for n sample and m evaluation points. The implementation is based on the method of Hofmeyr (2021), which uses kernels which can be expressed in the form

$$K(x) = \sum_{k=0}^{\alpha} \beta_k |x|^k \exp(-|x|), \tag{2}$$

for parameters $\beta_k > 0, k = 0, \ldots, \alpha$. This is simply a product of an arbitrary polynomial in $|x|$ with positive coefficients and the Laplace kernel $K(x) \propto e^{-|x|}$. The value α is referred to as the *order* of the kernel, as it represents the order of the polynomial component (i.e., the highest exponent). The advantage that these kernels have from a computational perspective is based on the fact that their sums allow for a recursive formulation in terms of the ordered observations, $x_{(1)}, \ldots, x_{(n)}$; $x_{(j)} \leq x_{(j+1)}$ for $j = 1, \ldots, n-1$, where ties may be broken arbitrarily. These recursive formulations arise from the trivial factorization,

$$\exp\left(-\frac{\left|x_{(j)} - x_{(j+1)}\right|}{h}\right) = \exp\left(\frac{x_{(j)} - x_{(j+1)}}{h}\right) = \exp\left(\frac{x_{(j)}}{h}\right)\exp\left(-\frac{x_{(j+1)}}{h}\right),$$

and by applying the binomial expansion to terms of the form $\left|x_{(j)} - x_{(j+i)}\right|^k$, i.e.,

$$\left|x_{(j)} - x_{(j+i)}\right|^k = (x_{(j+i)} - x_{(j)})^k = \sum_{l=0}^{k} \binom{k}{l} x_{(j+i)}^l (-x_{(j)})^{k-l}.$$

In particular, it can be shown that (Hofmeyr 2021)

$$S(x \mid \mathbf{x}, \omega) = \sum_{j=1}^{n} K\left(\frac{x_j - x}{h}\right)\omega_j = \sum_{k=0}^{\alpha} \beta_k \sum_{j=1}^{n} \frac{|x_j - x|^k}{h^k} \exp\left(-\frac{|x_j - x|}{h}\right)\omega_j = \tag{3}$$

$$\sum_{k=0}^{\alpha} \frac{\beta_k}{h^k} \sum_{l=0}^{k} \binom{k}{l} \left(\exp\left(\frac{x_{(n(x))} - x}{h}\right) x^{k-l} \ell(l, n(x)) + \exp\left(\frac{x - x_{(n(x))}}{h}\right)(-x)^{k-l} r(l, n(x))\right)$$

where $n(x)$ is the number of observations which are less than or equal to x, and the terms

$$\ell(l, j) := \sum_{i=1}^{j} (-x_{(i)})^l \exp\left(\frac{x_{(i)} - x_{(j)}}{h}\right)\omega_{(i)}; \tag{4}$$

$$r(l, j) := \sum_{i=j+1}^{n} x_{(i)}^l \exp\left(\frac{x_{(j)} - x_{(i)}}{h}\right)\omega_{(i)} \tag{5}$$

can easily be computed recursively in j for any fixed l. Here we have used $\omega_{(i)}$ to be the element in $\boldsymbol{\omega}$ associated with the i-th ordered element in the observations, $\mathbf{x}$. The important point here is that the evaluation of the expression in Equation 3 requires computational time which is independent of n, since the summation over the observations is shifted to the terms in Equations 4 and 5. Evaluation of Equation 3 for m evaluation points therefore has computational complexity $\mathcal{O}(m)$, while evaluating all of the terms of the form in Equations 4 and 5 has complexity $\mathcal{O}(n)$. To first compute the order of the observations and to determine

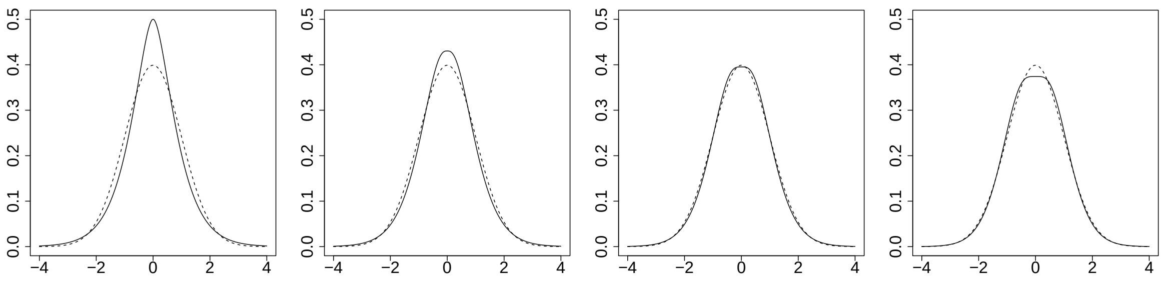


Figure 1: Smooth kernels of increasing order from one (left) to four (right). In addition the Gaussian kernel is shown (dashed) for comparison. The order one kernel, in the leftmost plot, is the default used in the package.

the values of $n(\cdot)$ for each evaluation point, however, jointly requires $\mathcal{O}(n\log(n) + m\log(m))$ time, which is thus the computational bottleneck in large problems.

The default kernel used in the package is an order one kernel with $\beta_0 = \beta_1 = \frac{1}{4}$. This is the simplest kernel of the form in Equation 2 with two continuous derivatives. To visualize the shape of kernels of the type in Equation 2, the package provides the function `plot_kernel(beta, ...)`. The function requires only a single argument, which is the vector of coefficients $(\beta_0, \dots, \beta_\alpha)$. In addition any graphical arguments accepted by R's base `plot()` function are accepted. The visualization produced by the function is scaled so that the kernel represents a probability density function of a random variable with unit variance. The normalizing constant can be determined using the function `norm_const_K(beta)`. That is, if `beta` is a vector of positive coefficients, then the kernel with coefficients `beta/norm_const_K(beta)` has unit integral. In addition `var_K(beta)` computes the variance of the random variable having as density the kernel with coefficients `beta/norm_const_K(beta)`. This standardization of scale allows for far simpler visual comparison of the different kernels available in this class. Hofmeyr (2021) discusses a sub-class of smooth kernels (a relatively high number of continuous derivatives at zero), for which $\beta_k \propto \frac{1}{k!}, k = 0, \dots, \alpha$. The first four of these are shown in Figure 1, with the popular Gaussian kernel, $K(x) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2}$, shown for comparison. The following code snippet can be used to reproduce this figure.

```
R> par(mfrow = c(1, 4), mar = rep(2, 4))
R> for (k in 1:4) {
+    plot_kernel(1 / factorial(0:k), ylim = c(0, 0.5))
+    lines(seq(-4, 4, length = 500),
+      dnorm(seq(-4, 4, length = 500)), lty = 2)
+  }
```

As can be seen in the figure, the order three kernel is visually similar to the Gaussian kernel. However, it is the order four kernel which has the closest efficiency to that of the Gaussian, for the purpose of density estimation. We have found that the default (order one) kernel is a very useful general purpose kernel, and have not often found reason to deviate from this choice.

Now, as we have discussed, of vital importance to the implementation of kernel-type estimators is the evaluation of sums of the form in Equation 1. Collections of such sums can be used for extremely flexible estimation of density functions, regression functions and spatial fields.

However, of potentially greater importance, in some applications, than the estimation of a function, is the estimation of its derivative. Examples include when the function values themselves are not of great consequence, but the structure of the function, in terms of its stationary points, is the focus of the analysis. Furthermore, in the context of projection pursuit, when the objective function is based on a kernel-type estimator of the distribution of the projected data, it is necessary to compute sums of kernel derivatives in order to evaluate the gradient of the objective during optimization. The **FKSUM** package therefore also offers functionality for evaluating such sums exactly and efficiently. Formally, the function `fk_sum()` may be used to evaluate the collection of sums

$$\sum_{i=1}^{n} K\left(\frac{x_i - \tilde{x}_j}{h}\right)\omega_i \text{ and } \sum_{i=1}^{n} K'\left(\frac{x_i - \tilde{x}_j}{h}\right)\omega_i, \text{ for } j = 1, \ldots, m, \tag{6}$$

where $(x_1, \ldots, x_n)$ is a vector of univariate sample points, and $(\tilde{x}_1, \ldots, \tilde{x}_m)$ is a vector of evaluation points. The function takes the following arguments:

`x`: Vector of sample points $(x_1, \ldots, x_n)$.

`omega`: Vector of coefficients $(\omega_1, \ldots, \omega_n)$.

`h`: Numeric bandwidth. Must be positive, i.e., `h > 0`.

`x_eval`: (Optional) vector of evaluation points $(\tilde{x}_1, \ldots, \tilde{x}_m)$. The default is `x`.

`beta`: (Optional) vector of kernel coefficients. The default is `c(0.25, 0.25)`, corresponding to the smooth order 1 kernel.

`nbin`: (Optional) integer number of bins if binned estimator is to be used. If omitted then exact evaluation is performed.

`type`: (Optional) one of `"ksum"`, `"dksum"` and `"both"`. If `"ksum"` or `"dksum"` then the first or second set of sums in Equation 6, respectively, is returned. If `"both"` then the matrix `cbind(ksum, dksum)` is returned. The default is `"ksum"`.

2.1. A simple application of `fk_sum()`

Before moving on to the main application covered in the paper in the following section, we introduce the reader to the functionality of the `fk_sum()` function through a simple example. Although the estimator we consider has an explicit implementation in the package, we believe it is beneficial to become familiar with the use of the general purpose function, for the purpose of modifying existing methods and implementing ones own. We will also cover a more complex example in a later section.

Example: Kernel regression

In the standard formulation of the regression problem, the mean of a response variable, Y, is related to one or more covariates, $X_1, \ldots, X_d$, through some function $f(\cdot)$, which needs to be estimated. That is,

$$Y = f(X_1, \ldots, X_d) + \epsilon,$$

where ϵ is a zero mean random variable, referred to as a *residual*, which may or may not be dependent on $X_1, \ldots, X_d$. Here we will assume that a single covariate, X, is available, and we observe a sample of pairs $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, assumed to be independent realizations from the joint distribution of X and Y. We will consider the simplest kernel-type estimator of the regression function, $f(\cdot)$, named after its two independent originators, the Nadaraya-Watson estimator (Nadaraya 1964; Watson 1964). The estimator is based on the very simple reasoning that to estimate $f(x) = E[Y \mid X = x]$ we can take a weighted average of the observed response values, $y_1, \ldots, y_n$, which emphasizes those for which the corresponding x_i's are close to x. Using kernels to provide these weights, we have

$$\hat{f}(x) = \frac{\sum_{i=1}^n K\left(\frac{x_i - x}{h}\right) y_i}{\sum_{i=1}^n K\left(\frac{x_i - x}{h}\right)}.$$

To compute $\hat{f}(\cdot)$ for a range of evaluation points, we evaluate the numerator and denominator terms each using the function `fk_sum()`. In the numerator, the vector of coefficients, $\boldsymbol{\omega}$, is given by the vector of responses $\mathbf{y} = (y_1, \ldots, y_n)$, while in the denominator $\boldsymbol{\omega}$ is given by a vector of ones.

Below we sample a set of observations and plot the kernel based estimates for two bandwidth values, one small and one large. We generate data in which the underlying regression function is given by $f(x) = 3\sin(2x) + 10(x-5)I(x > 5)$, where $I(\cdot)$ is the indicator function. This is a sine function with a kink at the point $x = 5$, above which a steep linear component is added. The residual distribution is equal to that of $T + (G-1)\left((X-5)^2 + 3\right)$, where T has a t distribution with 3 degrees of freedom and $G \sim \text{Gamma}(2, 2)$. We sample 5000 pairs where $\frac{1}{10}X \sim \text{Beta}(2, 2)$ and the distribution of $Y \mid X$ is described above.

```
R> set.seed(1)
R> n <- 5000
R> x <- rbeta(n, 2, 2) * 10
R> fx <- 3 * sin(2 * x) + 10 * (x > 5) * (x - 5)
R> y <- fx + rt(n, 3) + (rgamma(n, 2, 2) - 1) * ((x - 5)^2 + 3)
R> xeval <- seq(0, 10, length = 1000)
R> par(mfrow = c(1, 2))
R> for (h in c(0.25, 0.05)) {
+    plot(x, y, xlab = "x", ylab = "y")
+    fhat <- fk_sum(x, y, h, x_eval = xeval) / fk_sum(x, rep(1, n),
+      h, x_eval = xeval)
+    lines(xeval, fhat, col = 2, lwd = 2)
+    lines(xeval, 3 * sin(2 * xeval) + 10 * (xeval > 5) * (xeval - 5),
+      col = 3, lwd = 2)
+  }
```

The results are shown in Figure 2. The true function is shown in green, and the estimates in red. The larger bandwidth oversmooths and does not accurately capture the local extrema of the function (left). On the other hand, the smaller bandwidth is accurate over much of the range, but exhibits increased variation in the tails, which can be seen from what some authors describe as "wiggly" (right). The problem of bandwidth selection is beyond the scope of this paper, however for illustrative purposes we also plot estimates using two data

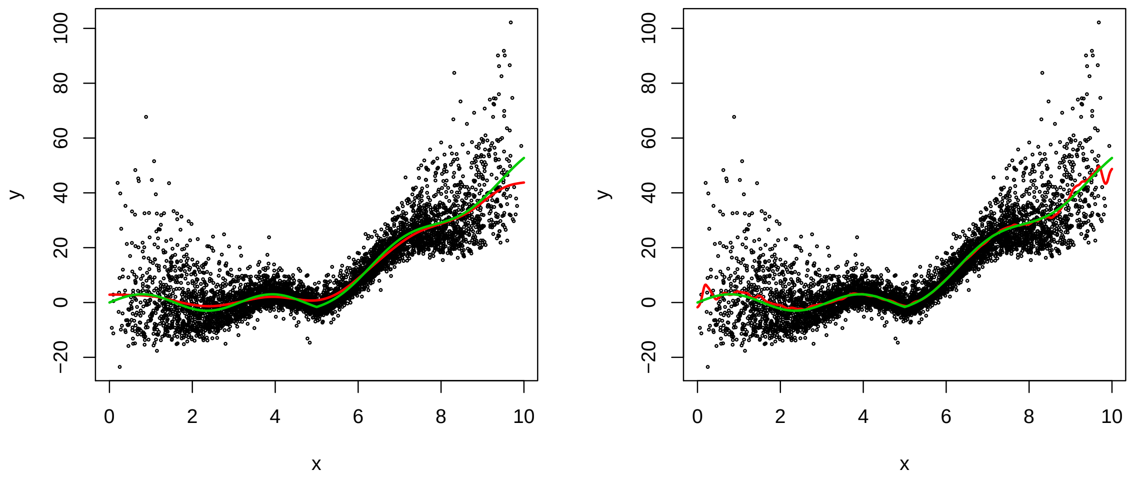


Figure 2: Nadaraya-Watson regression estimates of non-linear function. True function shown in green, with estimates using bandwidth 0.25 (left) and 0.05 (right) shown in red.

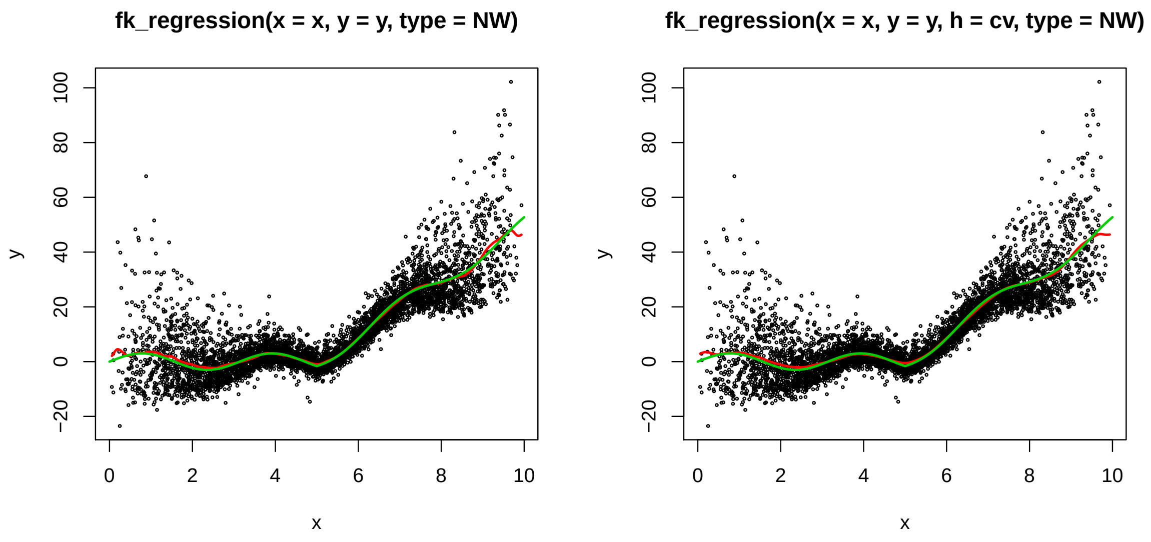


Figure 3: Nadaraya-Watson regression estimates of non-linear function. True function shown in green, with estimates using amise bandwidth (left) and cross validation based bandwidth (right) shown in red

driven bandwidth selection techniques implemented in the package. The first is based on an approximation of the asymptotic mean integrated squared error minimizing bandwidth (amise), while the other is based on leave-one-out cross validation (cv). For these we use the purpose built function `fk_regression()`.

```
R> par(mfrow = c(1, 2))
R> plot(fk_regression(x, y, type = "NW"))
R> lines(xeval, 3 * sin(2 * xeval) + 10 * (xeval > 5) * (xeval - 5),
+    col = 3, lwd = 2)
```

```
R> plot(fk_regression(x, y, type = "NW", h = "cv"))
R> lines(xeval, 3 * sin(2 * xeval) + 10 * (xeval > 5) * (xeval - 5),
+    col = 3, lwd = 2)
```

The argument `type` specifies the use of the Nadaraya-Watson estimator, where the default would be a local-linear estimator. The output can be seen in Figure 3. Both bandwidth values lead to an estimate which captures the function well with considerably less variation than the smaller bandwidth from the first set of estimates.

In this section we introduced the versatile function `fk_sum()`, which can be used to implement standard kernel based estimators very simply. The example covered should provide the reader with sufficient understanding of the function's use that they will be capable of implementing their own simple kernel estimators. In the next section we discuss some of the functionality offered by the package for projection pursuit problems. In addition to two purpose-built implementations, we discuss in detail the implementation of projection pursuit regression using the Nadaraya-Watson estimator. This example should give more advanced readers further instruction on the use of the function `fk_sum()` in more complex problems.

3. Projection pursuit

Projection pursuit refers to the problem of identifying (usually low-dimensional) linear projections of a set of data which expose structures of interest. What is considered interesting may be made explicit in relation to a subsequent objective, e.g., clustering (Bolton and Krzanowski 2003; Hofmeyr and Pavlidis 2019), or the pursuit may be more exploratory in nature (Huber 1985; Friedman 1987). Projection pursuit may also be formulated for supervized problems, such as regression (Friedman and Stuetzle 1981), in which projections of a collection of covariates are sought which reveal strong predictive relationships with a set of response variables. Arguably the most popular projection pursuit method is that of Principal Component Analysis (PCA). PCA has numerous equivalent formulations, including finding the projection of the data which minimizes the total squared reconstruction error. In the context of projection pursuit this may be seen rather as the objective of finding projections which lose as little structure as possible, in a general sense. However, PCA may fail in this objective if the variation in the noisy (less structured) components in the data dominates the total data variation.

The basic formulation of the projection pursuit problem may be given as

$$\max_{\mathbf{W} \in \mathcal{F}} \Phi(\mathbf{W} \mid \mathbf{X}),$$

where $\mathbf{X} \in \mathbb{R}^{n \times d}$ represents the data matrix, $\mathbf{W}$ is the *projection matrix*, which comes from some feasible set $\mathcal{F} \subset \mathbb{R}^{d \times d'}$, where d' is the dimension of the projection; and $\Phi(\cdot)$ is the *projection index*, which is intended to measure how "interesting" the projected data are. That is, we seek the projection matrix, $\mathbf{W}$, for which the projected data, $\mathbf{XW} \in \mathbb{R}^{n \times d'}$, capture the structures of interest (which are in $\mathbf{X}$) as well as possible. Very frequently an equivalent formulation of the projection index allows us to decompose the total "interestingness" of the projected data into the sum of the interestingness of each of the *components*, $\mathbf{X}\mathbf{w}_i, i = 1, \dots, d'$, measured separately. Even if the reformulation is not equivalent, this latter approach is frequently preferred as it can vastly simplify the problem. The main practical benefit of this reformulation is that it allows us to obtain the *projection vectors*, $\mathbf{w}_i, i = 1, \dots, d'$ (i.e., the columns of $\mathbf{W}$), iteratively. The feasible set $\mathcal{F}$ is then used (either explicitly or implicitly) to effectively ensure that the same, or very similar vectors are not obtained for multiple columns of $\mathbf{W}$. With this reformulation we consider the projection pursuit problem expressed, somewhat loosely, as,

$$\max_{\mathbf{W}:\mathbf{w}_i \in \mathcal{F}(\mathbf{W}_{-i})} \sum_{i=1}^{d'} \Phi\left(\mathbf{w}_i \mid \mathbf{X}, \mathbf{W}_{-i}\right),$$

where $\mathbf{W}_{-i}$ denotes the matrix $[\mathbf{w}_1 \; \dots \; \mathbf{w}_{i-1} \; \; \mathbf{w}_{i+1} \; \dots \; \mathbf{w}_{d'}]$, i.e., the projection matrix excluding the i-th column. That is, we make explicit the fact that the feasible set, and indeed the projection index itself, for the i-th component, may depend on the values of the other components. As mentioned above, the problem in this form is often approached in an iterative manner, in which the optimization is greedily performed by adding the apparently best component at iteration i, given all those obtained so far, but without accounting for components which will be obtained subsequently. In this case we are essentially then interested only in the univariate projection pursuit problem,

$$\max_{\mathbf{w} \in \mathcal{F}} \Phi(\mathbf{w} \mid \mathbf{X}),$$

where we have suppressed the dependence on the other components for convenience of notation, and note that in order to obtain a complete solution we may have to consider a sequence of different problems, i.e., with different objectives and feasible sets. Now, except in examples in which the projection index is *scale invariant*, i.e., $\Phi(\mathbf{w} \mid \mathbf{X}) = \Phi(\alpha\mathbf{w} \mid \mathbf{X}) \; \forall \alpha > 0$, a fairly universal constraint is that the norm of the projection vector is constant, where without loss of generality we may assume $||\mathbf{w}|| = 1$. Enforcing this constraint can be achieved in a number of ways, including using modified search directions during optimization (Niu, Dy, and Jordan 2011), expressing $\mathbf{w}$ in polar coordinates (Hofmeyr and Pavlidis 2015) or by formulating the projection index as the composition (Hofmeyr 2017)

$$\Phi(\mathbf{w} \mid \mathbf{X}) = \phi(\mathbf{p})\Big|_{\mathbf{p}=\mathbf{X}\vec{\mathbf{w}}},$$

where $\vec{\mathbf{w}} = \frac{1}{||\mathbf{w}||}\mathbf{w}$ and $\phi(\cdot)$ simply measures the interestingness of a univariate data set, here given as the normalized projected data, stored in the vector $\mathbf{p} = (p_1, \dots, p_n) = (\mathbf{x}_1^\top \vec{\mathbf{w}}, \dots, \mathbf{x}_n^\top \vec{\mathbf{w}})$. Practically, then, $\Phi(\cdot)$ preforms two operations. First it projects the argument $\mathbf{w}$ onto the unit ball (i.e., divides it by its norm), and then computes the interestingness of the data projected onto the unit vector $\vec{\mathbf{w}}$. Optimization of $\Phi(\cdot)$ based on this formulation can thus equivalently be performed without any constraint on the magnitude of the argument $\mathbf{w}$.

There are three projection pursuit methods which are implemented in **FKSUM**. These are independent component analysis (ICA, Hyvärinen and Oja 2000), minimum density projection pursuit (MDPP, Pavlidis *et al.* 2016) and projection pursuit regression (PPR, Friedman and Stuetzle 1981). We will discuss the purpose-built implementations of ICA and MDPP in this section. In addition, we provide a detailed discussion of the implementation of projection pursuit regression with the intention that this will provide instruction for readers who may wish to implement their own projection pursuit methods, or existing methods which are not implemented in the package.

3.1. Independent component analysis

A primary motivation for the application of independent component analysis is the assumption that an observed multivariate data set, $\mathbf{X} \in \mathbb{R}^d$, has as columns different linear combinations of latent factors which are statistically independent. The task is then to obtain an *unmixing matrix*, $\mathbf{W}$, such that the columns of the transformed data $\mathbf{XW}$ have maximal independence. This has broad application in signal processing; with the canonical example being the so-called *cocktail party problem.* Here a room is populated by a crowd of people, with different subsets involved in different conversations. Microphones situated around the room record these conversations, with each microphone recording conversations closer to them with higher volume, but also receiving sounds from all other conversations in the room at different levels. The objective is then to take all recordings and identify a transformation which separates the actual conversations from one another.

A popular objective for ICA uncovers an alternative point of view which motivates the application of ICA to the general problem of exploratory projection pursuit. That is, to maximize independence one can minimize the mutual information in the components of the projected data. In order to achieve this, we attempt to maximize the unique information in each of the components, and so in a general sense ICA may be seen as obtaining the projection which identifies the most informative components in the data. Now, if we consider the statistical context, in which our observations (stored in the rows of $\mathbf{X}$) represent a sample of realizations of a random variable, say $\mathbf{Z}$, which may without loss of generality be assumed to have zero mean, then the objective of minimizing mutual information is to minimize the Kullback-Leibler divergence (KL divergence, Kullback and Leibler 1951) of the product of the marginal densities of the components of $\mathbf{W}^\top\mathbf{Z}$ from their joint density. That is, if we use $f_{\mathbf{Z}}(\cdot)$, $f_{\mathbf{W}^\top\mathbf{Z}}(\cdot)$ and $f_{\mathbf{w}_i^\top\mathbf{Z}}(\cdot)$ to represent the densities of $\mathbf{Z}$, $\mathbf{W}^\top\mathbf{Z}$ and $\mathbf{w}_i^\top\mathbf{Z}$ respectively, then the mutual information in the components of $\mathbf{W}^\top\mathbf{Z}$ is given by

$$\begin{aligned} KL\left(f_{\mathbf{W}^\top\mathbf{Z}}\middle\| \prod_{i=1}^{d'} f_{\mathbf{w}_i^\top\mathbf{Z}}\right) &= E\left[\log(f_{\mathbf{W}^\top\mathbf{Z}}(\mathbf{W}^\top\mathbf{Z}))\right] - E\left[\log\left(\prod_{i=1}^{d'} f_{\mathbf{w}_i^\top\mathbf{Z}}(\mathbf{w}_i^\top\mathbf{Z})\right)\right] \\ &= E[f_{\mathbf{Z}}(\mathbf{Z})] + \log(\det|\mathbf{W}|) - \sum_{i=1}^{d'} E[\log(f_{\mathbf{w}_i^\top\mathbf{Z}}(\mathbf{w}_i^\top\mathbf{Z}))]. \end{aligned}$$

If this divergence is small, then the joint density is well approximated by the product of its marginals, meaning that the corresponding random variables are close to independent; sharing little information between them. Now, two useful observations vastly simplify the problem of estimating an appropriate projection matrix, $\mathbf{W}$, to achieve this minimization. First, a necessary condition for two random variables to be independent is that they have zero correlation. Second, independence is unaffected by scale, in that two random variables, say U, V are independent if and only if U and αV are independent for all α. Practically, then, we restrict $\mathbf{W}$ to be of the form $\mathbf{\Lambda}^{-1/2}\mathbf{UQ}$, where $\mathbf{\Lambda}$ is the diagonal matrix containing the eigenvalues of the covariance of $\mathbf{Z}$, $\mathbf{U}$ has as columns its first d' eigenvectors, and $\mathbf{Q} \in \mathbb{R}^{d'\times d'}$ is orthonormal. The convenience of this is that the determinant of $\mathbf{W}$ is constant, meaning that the dependence of the above KL-divergence on $\mathbf{W}$ is only in the term $\sum_{i=1}^{d'} E[\log(f_{\mathbf{w}_i^\top\mathbf{Z}}(\mathbf{w}_i^\top\mathbf{Z}))]$. This is the sum of the negative entropies of the random variables $\mathbf{w}_1^\top\mathbf{Z}, \dots, \mathbf{w}_{d'}^\top\mathbf{Z}$. Notice also that the term $\mathbf{\Lambda}^{-1/2}\mathbf{U}$ has the effect of whitening the random variable, so that $\mathbf{Z\Lambda}^{-1/2}\mathbf{U}$ has identity covariance. Practically, then, to perform ICA, we first whiten the data matrix by setting $\tilde{\mathbf{X}} = (\mathbf{X} - \hat{\boldsymbol{\mu}}\mathbf{1}^\top)\hat{\boldsymbol{\Lambda}}^{-1/2}\hat{\mathbf{U}}$, where $\hat{\boldsymbol{\Lambda}}$ and $\hat{\mathbf{U}}$ contain the eigenvalues and vectors of the covariance matrix of the data, and $\hat{\boldsymbol{\mu}}$ is the vector of column means of the data. We then iteratively minimize the estimated sample entropy of the projected components, with the i-th minimization being of

$$-\frac{1}{n}\sum_{j=1}^{n}\log\left(\hat{f}_{\mathbf{p}}(p_j)\right)\Big|_{\mathbf{p}=\tilde{\mathbf{X}}\vec{\mathbf{q}_i}},$$

over the projection vector, now denoted by $\mathbf{q}_i$, under the constraint that $\mathbf{q}_i$ is orthogonal to $\mathbf{q}_1, \dots, \mathbf{q}_{i-1}$. Here $\mathbf{q}_i$ is, after normalization, the i-th column of the orthonormal matrix $\mathbf{Q}$. We use $\hat{f}_{\mathbf{p}}(\cdot)$ to be the estimated density based on the vector of univariate projected points, where in this case this is given by the projection of the whitened data, $\tilde{\mathbf{X}}$, onto the normalized projection vector $\vec{\mathbf{q}_i}$. The natural choice of density estimate in our context is the kernel density estimate, given by

$$\hat{f}_{\mathbf{p}}(x) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{p_i - x}{h}\right).$$

The implementation of ICA in **FKSUM** is provided in the function `fk_ICA()`, and is based on the above approach. The function takes the following arguments:

`X`: Matrix of observations $\mathbf{X}$ (num_data $\times$ num_variables).

`ncomp`: (Optional) integer number of independent components to extract. The default is `1`.

`beta`: (Optional) vector of kernel coefficients. The default is `c(0.25, 0.25)`.

`hmult`: (Optional) bandwidth multiplier. Bandwidth used is Silverman's rule (Silverman 1986) multiplied by `hmult`. The default is `1.5`.

`it`: (Optional) integer number of iterations in each optimization. The default is `20`.

`nbin`: (Optional) integer number of bins if binning approximation is to be used. The default is to perform exact kernel computations.

The output of the function is a named list with class `fk_ICA`, containing the following fields:

`$X`: The data matrix given as argument to the function.

`$K`: The whitening matrix, $\hat{\boldsymbol{\Lambda}}^{-1/2}\hat{\mathbf{U}}$.

`$W`: The optimal projection matrix for the whitened data.

`$S`: The estimated independent components.

Note that in the package we denote the unmixing matrix for the already whitened data by `$W` to be consistent with other implementations, where in our derivation above we used $\mathbf{Q}$ to represent this matrix.

3.2. Examples

Here we use the function `fk_ICA()` to perform independent component analysis. We will consider both simulated and real data examples in order to illustrate the speed and accuracy of the package implementation. For context, we compare with the ICA methods implemented in the packages **fastICA** (Marchini *et al.* 2021), **ProDenICA** (Hastie and Tibshirani 2010), **PearsonICA** (Karvanen and Koivunen 2002) and **JADE** (Miettinen *et al.* 2017).

Simulation

In the first example using the `fk_ICA()` function we conduct a simulation study to assess the speed and accuracy of the implementation in recovering components which are simulated under the independence assumption, but which are then subjected to a random linear transformation using a randomly generated mixing matrix. Following Bach and Jordan (2002) and Hastie and Tibshirani (2003), we use the Amari distance (Amari, Cichocki, and Yang 1996) between the estimated unmixing matrix and the inverse of the mixing matrix to assess the accuracy of the estimation. We also use the collection of densities introduced by Bach and Jordan (2002) to simulate the true independent components.

The **ProDenICA** package also conveniently provides functions for simulating data from the densities in Bach and Jordan (2002) and for computing the Amari distance, in the functions `rjordan()` and `amari()` respectively. It also provides the function `mixmat()` for simulating mixing matrices.

We repeatedly generate sets of data with independent components, and then apply a randomly generated linear transformation to these data and apply the various ICA methods. We repeat the experiment 20 times, and in each case sample 2000 points with 4 components. Within each experiment we choose a random combination of the 18 densities produced by the `rjordan()` function, which are listed according to the letters `'a'` to `'r'`. For illustration, the following produces a single such data set and applies the function `fk_ICA()` to estimate the independent components. The matrix `X` contains the true components and `R` is used to represent the random mixing matrix. The matrix `Xx` therefore contains the mixed components.

```
R> set.seed(1)
R> X <- matrix(0, 2000, 4)
R> densities <- sample(letters[1:18], 4)
R> for (i in 1:4)  X[,i] <- rjordan(densities[i], 2000)
R> R <- mixmat(4)
R> Xx <- X %*% R
R> model <- fk_ICA(Xx, 4)
```

Next we plot the densities of the true components, the mixtures and the estimated components.

```
R> par(mfrow = c(4, 3), mar = c(0, 0, 3, 0))
R> for (i in 1:4) {
+    plot(fk_density(Xx[,i]), main = paste("Mixed component", i),
+      xaxt = "n", yaxt = "n")
+    plot(fk_density(model$S[,i]), main = paste("Estimated component", i),
+      xaxt = "n", yaxt = "n")
```

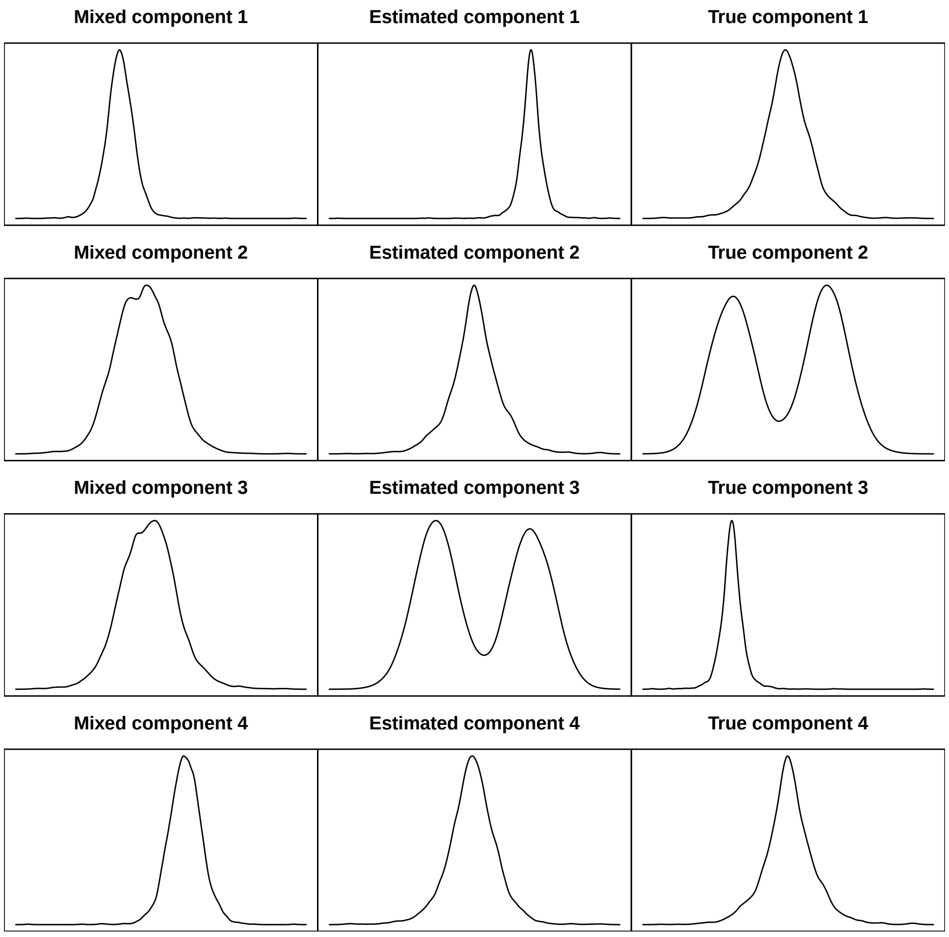


Figure 4: Mixed, estimated and true components from simulated data.

```
+     plot(fk_density(X[,i]), main = paste("True component", i),
+       xaxt = "n", yaxt = "n")
+   }
```

The plots are shown in Figure 4. Note that the order of the estimated components does not necessarily coincide with that of the true components. The ICA objective is also invariant to reflection, and hence some components may be reflected. It is clearly verifiable by eye that the estimated components accurately capture the true components.

The results from all 20 replications of this experiment are summarized in Figure 5 below. Code to reproduce these results is available in supplementary material. In addition to the implementation based on exact kernel estimation of the component entropies, we have also considered a binned approximation using the optional argument `nbin` in the function `fk_ICA()`. This offers a more relevant comparison with the method in **ProDenICA** which uses a similar approximation technique. The Amari distances are shown in the boxplots in Figure 5, where the mean distance for each method is also included. In addition the horizontal line corresponds with the median Amari distance from the default implementation of `fk_ICA()`, which allows for easier comparisons.

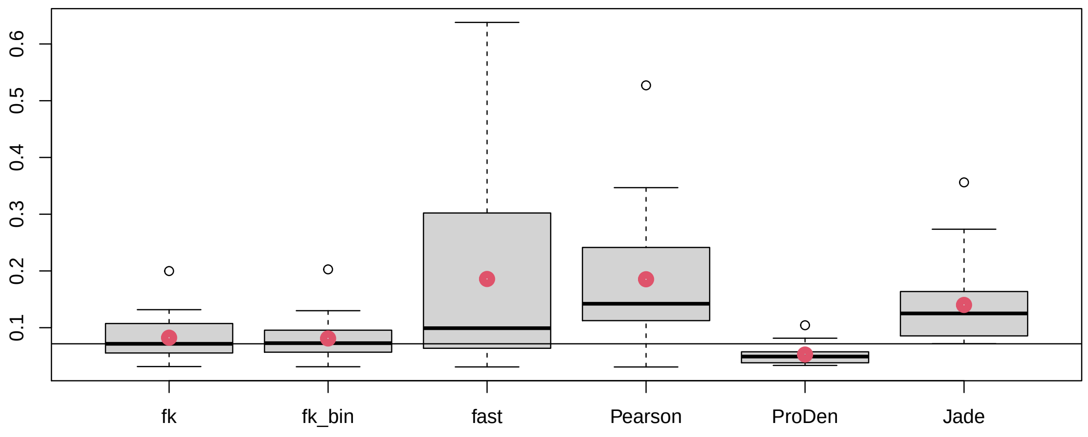


Figure 5: Boxplots of Amari distances between estimated unmixing matrix and inverse mixing matrix. In addition the mean Amari distance for each method is shown. The horizontal line corresponds to the median for the default settings of `fk_ICA()`.

The implementation in **ProDenICA** obtains the most accurate results, but runs an order of magnitude slower than any of the other methods, taking an average of 2 seconds per run compared with `fk_ICA()` taking an average of 0.15 seconds without and 0.09 seconds with binning. The running time will clearly depend on the system being used, but we expect a similar relative running time for comparison on all systems. The implementations in **fastICA**, **PearsonICA** and **JADE** are very computationally efficient, but they are much less reliable in accurately recovering the underlying components. The method in **FKSUM** offers a reasonable trade-off; achieving close-to as accurate outputs to those of **ProDenICA**, but far more efficiently, on these data.

The cocktail party problem

Here we follow Example 1 of Miettinen *et al.* (2017), but repeat it multiple times. The example uses three audio waves of length 50000 and a single noise signal as the original components, and then applies a random mixing matrix to obtain the observed mixed signals. We begin by loading the package required for this example, and then loading the audio data.

```
R> library("tuneR")
R> S1 <- readWave(system.file("datafiles/source5.wav", package = "JADE"))
R> S2 <- readWave(system.file("datafiles/source7.wav", package = "JADE"))
R> S3 <- readWave(system.file("datafiles/source9.wav", package = "JADE"))
```

Next we repeatedly generate a random noise signal and mixing matrix, and fit the same models as in the last example. We compute the amari distance between the estimated unmixing matrices and the inverse of the mixing matrix, `R`, and store these in the list `amari_all`. We also store the running times of the different methods in the list `t_all`. We repeat the experiment 20 times.

```
R> amari_all <- list(fk = c(), fk_bin = c(), fast = c(),
+    Pearson = c(), ProDen = c(), Jade = c())
```

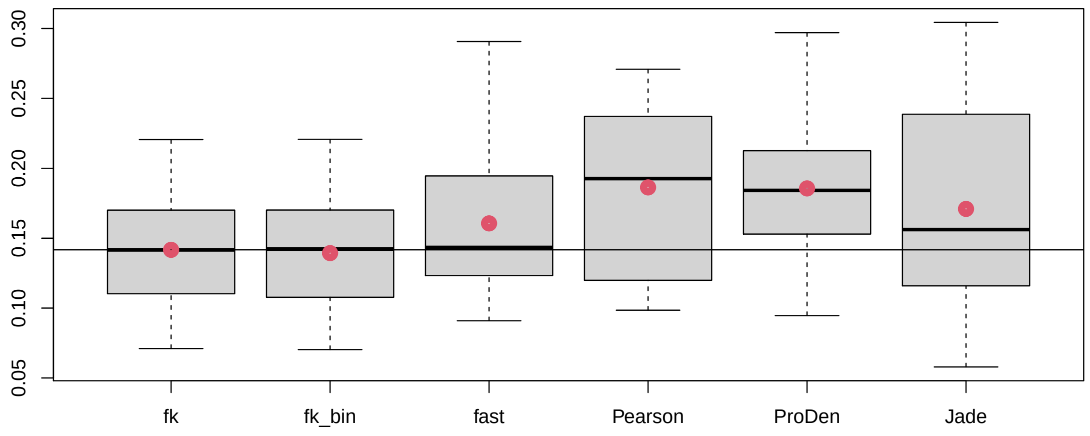


Figure 6: Boxplots of Amari distances between estimated unmixing matrix and inverse mixing matrix. In addition the mean Amari distance for each method is shown. The horizontal line corresponds to the median for the default settings of `fk_ICA()`.

```
R> t_all <- list(fk = c(), fk_bin = c(), fast = c(),
+    Pearson = c(), ProDen = c(), Jade = c())
R> for (rep in 1:20) {
+    set.seed(rep)
+    NOISE <- noise("white", duration = 50000)
+    X <- cbind(S1@left, S2@left, S3@left, NOISE@left)
+    R <- mixmat(4)
+    Xx <- X %*% R
+    t_all$fk[rep] <- system.time(model <- fk_ICA(Xx, 4))[1]
+    amari_all$fk[rep] <- amari(model$K %*% model$W, solve(R))
+    t_all$fk_bin[rep] <- system.time(model <- fk_ICA(Xx, 4, nbin = 5000))[1]
+    amari_all$fk_bin[rep] <- amari(model$K %*% model$W, solve(R))
+    t_all$fast[rep] <- system.time(model <- fastICA(Xx, 4))[1]
+    amari_all$fast[rep] <- amari(model$K %*% model$W, solve(R))
+    t_all$ProDen[rep] <- system.time(model <- ProDenICA(Xx, 4,
+      whiten = TRUE))[1]
+    amari_all$ProDen[rep] <- amari(model$whitener %*% model$W, solve(R))
+    t_all$Pearson[rep] <- system.time(model <- PearsonICA(Xx, 4))[1]
+    amari_all$Pearson[rep] <- amari(model$W, solve(R))
+    t_all$Jade[rep] <- system.time(model <- JADE(Xx))[1]
+    amari_all$Jade[rep] <- amari(t(model$W), solve(R))
+  }
```

As before we plot the boxplots of the Amari distances, and add the mean Amari distance for each method. We also add a horizontal line at the median of the performance from the method in **FKSUM** to make the comparisons clearer.

```
R> boxplot(amari_all)
R> abline(h = median(amari_all$fk))
R> points(1:6, unlist(lapply(amari_all, mean)), col = 2, lwd = 6)
```

The outputs are shown in Figure 6. In this example `fk_ICA()` obtains the most accurate results, with only minor differences between the exact kernel estimates and the binned approximation. In fact the binned approximation yields a very slightly more accurate output than the exact kernel method. Unlike in the simulation example covered previously, `ProDenICA()` is one of the least accurate methods.

The average running times tell a similar story to those from the previous example.

The main difference here is that the binning approximations are relatively faster in this larger example. In particular, the binned approximation implemented in **FKSUM** is now reasonably competitive with the methods which use surrogate functions for the negative entropy, running roughly 5 times slower. In addition, the binned implementation in **ProDenICA** now runs in a similar amount of time to the exact method in **FKSUM**.

3.3. Minimum density hyperplanes

The minimum density projection pursuit method (Pavlidis *et al.* 2016) attempts to obtain the cluster separating hyperplane with minimal integrated density along it; the minimum density hyperplane (MDH). That is, to obtain the hyperplane $H(\mathbf{w}, b) := \{\mathbf{x} \in \mathbb{R}^d \mid \mathbf{w}^\top \mathbf{x} = b\}$ for which the surface integral on the hyperplane, $\oint_{H(\mathbf{w},b)} f(\mathbf{x})d\mathbf{x}$, is minimized. Here $f(\cdot)$ is a probability density function, and this integral is, in practice, estimated using a kernel estimate based on a sample from the distribution with density $f(\cdot)$. Notice that if X is a random variable with density $f(\cdot)$, then $\oint_{H(\mathbf{w},b)} f(\mathbf{x})d\mathbf{x}$ is equal to the value of the density of the univariate random variable $X^\top \vec{\mathbf{w}}$ evaluated at $\frac{b}{||\mathbf{w}||}$. To estimate this integral, therefore, one needs only compute a univariate density estimate from the projected data on $\vec{\mathbf{w}}$. Now, it should be clear that the integral on the hyperplane $H(\mathbf{w}, b)$ can be made arbitrarily small by taking a value of b which is extremely large in magnitude. To ensure the hyperplane passes through the region of the density which is of interest, rather than through the tail, a penalty is added to the objective which ensures the hyperplane passes within a chosen distance of the mean. In particular, the objective is given by

$$p(\mathbf{w}, b) = \widehat{I(\mathbf{w}, b)} + C \max\{0, |b - \hat{\mu}_{\mathbf{w}}| - \alpha\hat{\sigma}_{\mathbf{w}}\}^2, \tag{7}$$

where $\widehat{I(\mathbf{w}, b)}$ is used to represent the estimate of $\oint_{H(\mathbf{w},b)} f(\mathbf{x})d\mathbf{x}$ and the terms $\hat{\mu}_{\mathbf{w}} = \mathbf{w}^\top \hat{\boldsymbol{\mu}}$ and $\hat{\sigma}_{\mathbf{w}} = \sqrt{\mathbf{w}^\top \hat{\Sigma}\mathbf{w}}$ are the mean and standard deviation of the projected data, $\mathbf{X}\mathbf{w}$, respectively. The constant $C > 0$ is some large positive value which affects the strength of the penalty and $\alpha > 0$ controls the size of the feasible region. The second term in Equation 7 therefore applies no penalty for hyperplanes which pass within α standard deviations of the mean of the data, measured in the the direction orthogonal to the hyperplane. Beyond the distance $\alpha\hat{\sigma}_{\mathbf{w}}$, the penalty scales quadratically with the distance of the hyperplane from the mean. Especially when the number of clusters is large, the number of local minima in the objective in Equation 7 tends to be large. To mitigate the effect of these minima, the minimum density projection pursuit (MDPP) objective is given by,

$$\Phi(\mathbf{w} \mid \mathbf{X}) := \min_{b \in \mathbb{R}} p(\vec{\mathbf{w}}, b).$$

Practically, for a given $\mathbf{w}$, we compute a kernel density estimate from the projected data on the normalized projection vector, $\mathbf{X}\vec{\mathbf{w}}$, and search for the minimum of this density plus the penalty described in Equation 7. In other words, the projection index for projection vector $\mathbf{w}$ is the value of the original objective for the best hyperplane orthogonal to $\mathbf{w}$. This approach allows the value of b to change discontinuously during optimization, and thereby avoid some of the local minima in the original objective. The hyper-parameter α is adjusted during optimization. Starting from $\alpha = 0$, the first solution passes through the mean of the data, and tends to lead to a reasonable separation of clusters. Thereafter the value is slowly increased and the solution returned by the algorithm is the last *valid* cluster separator (a hyperplane which passes between the modes of the density of the projected data).

MDPP is implemented in the function `fk_mdh()`, which takes the following arguments:

`X`: Matrix of observations $\mathbf{X}$ (num_data $\times$ num_variables).

`v0`: (Optional) initial projection vector. The default is the first principal component.

`hmult`: (Optional) numeric bandwidth multiplier. The bandwidth used to estimate the density is given by `hmult` multiplied by Silverman's heuristic calculated on the initial projection vector. The default value is `1`.

`beta`: (Optional) vector of kernel coefficients. The default is `c(0.25, 0.25)`, corresponding to the smooth order 1 kernel.

`alphamax`: (Optional) numeric maximum (scaled) distance of the hyperplane from the mean of the data. The default is `1`.

The function returns a named list with class `fk_mdh`, and fields `$v` and `$b`, corresponding to the final (normalized) projection vector, $\vec{\mathbf{w}}$, and the value of b in the optimal hyperplane orthogonal to $\vec{\mathbf{w}}$.

It is worth noting that in order to compute $\Phi(\mathbf{w} \mid \mathbf{X})$, the search for the optimal hyperplane orthogonal to $\mathbf{w}$ is performed using a combination of grid evaluations and binary search. As a result, there is no need to evaluate the density at every projected data point, as was the case in ICA. We therefore do not expect so significant an improvement on the running time over existing implementations. That being said, repeated evaluation of the density during binary search is more efficient using the fast kernel computations availed by the package.

3.4. Examples

Simulation

We begin, as before, with a simulation to assess the speed and accuracy of the implementation in **FKSUM**. We compare with the implementation given in **PPCI** (Hofmeyr and Pavlidis 2019), which uses the Gaussian kernel for estimating densities. In each experiment we simulate 2000 data from a Gaussian mixture in 10 dimensions, with means and diagonal covariances determined randomly. The number of components in each mixture is also determined randomly. We measure the accuracy in terms of the true integrated density on the hyperplane solutions, as well as the clustering accuracy of the separation of the collections of points generated from each of the mixture components. Since a hyperplane only induces a binary partition of the data, we evaluate the clustering accuracy using the success ratio (Pavlidis *et al.* 2016), which measures the degree to which at least one cluster has been successfully separated from the

	Running time	Integrated density	Success ratio
FKSUM	0.116 ± 0.041s	0.078 ± 0.098	0.941 ± 0.095
PPCI	0.603 ± 0.314s	0.071 ± 0.096	0.948 ± 0.084

Table 1: Average ± standard deviation of running time, integrated density and success ratio for two implementations of MDPP.

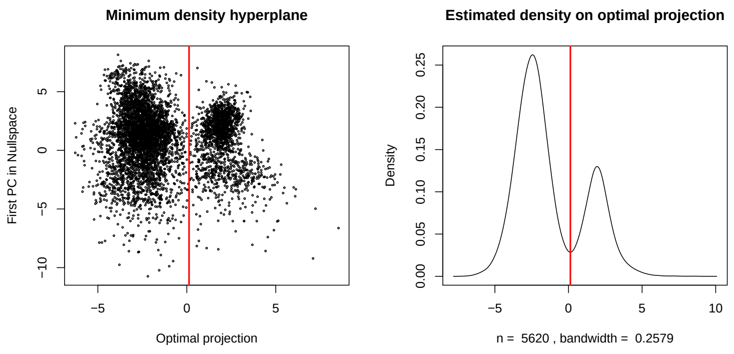
remainder. We repeat the experiment 100 times, and store the resulting running times and accuracy measures. The results are summarized in Table 1. Code to replicate these results can be found in supplementary materials. The only appreciable difference in the performance is in terms of running time, where the implementation in **FKSUM** runs an average of five times faster than the existing implementation.

Image clustering

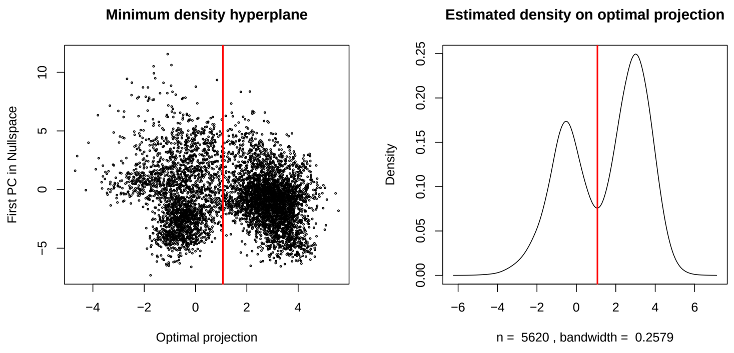
Next we apply both implementations of MDPP to the optical recognition of handwritten digits data set, which is available in the **PPCI** package, and which was originally obtained from the UCI machine learning repository (Dua and Karra Taniskidou 2017). The data represent 5620 images of handwritten digits from multiple subjects, which have been compressed to 8×8 pixels and vectorized, resulting in 64 dimensional data. We load this data set and apply both implementations.

```
R> data("optidigits", package = "PPCI")
R> system.time(fk_sol <- fk_mdh(optidigits$x))

   user  system elapsed
  3.689   0.057   3.746

R> success_ratio(optidigits$x %*% fk_sol$v < fk_sol$b[1], optidigits$c)

[1] 0.9299176

R> system.time(pp_sol <- mdh(optidigits$x))

   user  system elapsed
  5.318   0.157   5.476

R> success_ratio(optidigits$x %*% pp_sol$v < pp_sol$b[1], optidigits$c)

[1] 0.9040637
```

In this case the method in **FKSUM** offers only a minor improvement in running time, but obtains a superior clustering result. Next we plot the resulting hyperplanes, and the estimated density along the optimal projection vectors. The class `fk_mdh` includes the method `plot`. To utilize this method to visualize the solution from **PPCI**, we replace the projection vector and split point from the **FKSUM** solution.



(a) **FKSUM** solution



(b) **PPCI** solution

Figure 7: Minimum density hyperplane solutions on optical recognition of handwritten digits data. The left plots show the data projected into a two-dimensional subspace determined by the optimal projection vector and the first principal component in its null-space, while the right plots show the density of the data projected on the optimal projection vector.

```
R> plot(fk_sol)
R> fk_sol$v <- pp_sol$v
R> fk_sol$b <- pp_sol$b
R> plot(fk_sol)
```

The plots are shown in Figure 7. The left plots show scatter plots of the data projected into two-dimensional subspaces. In each case the horizontal axis corresponds to the optimal projection obtained from MDPP, while the vertical axis is the first principal component of the data after projection into the null-space of the MDPP projection. The right plots show the kernel density estimates computed from the data projected on the optimal projection vectors. It is apparent that the **FKSUM** method obtained a hyperplane of considerably lower density in this example.

3.5. Projection pursuit regression: A detailed implementation

In this final discussion of projection pursuit we provide a very detailed description of the implementation of projection pursuit regression (PPR). The intention is that the reader will find sufficient instruction for the implementation of their own projection indices, or indices which are not currently offered in the **FKSUM** package. For details on the use of the PPR implementation in **FKSUM**, use the command `help(fk_ppr)`.

The standard PPR model assumes the conditional distribution of the response variable, Y, given a vector of covariates, X, is given by

$$Y \mid X \sim N\left(\mu + \sum_{j=1}^{k} f_j(\mathbf{w}_j^\top X), \sigma^2\right),$$

where the functions $f_j : \mathbb{R} \to \mathbb{R}; j = 1, \ldots, k$, are assumed to come from some known class of functions, say $\mathcal{C}$. It is these functions, as well as the projection vectors $\mathbf{w}_1, \ldots, \mathbf{w}_k$ and the global mean, μ, which are to be estimated. Notice that if $\mathcal{C}$ is the class of linear functions, then the above problem reverts to the standard linear regression model. In general $\mathcal{C}$ is assumed to be a very rich class of functions, where restrictions might only be placed on the number of continuous derivatives admissible by its members. In these general cases nonparametric estimation of $f_j; j = 1, \ldots, k$ becomes preferable. The simplest approach to fitting a PPR model follows a forward procedure in which each term in the expression for the conditional expectation of the response,

$$E[Y \mid X = \mathbf{x}] = \mu + \sum_{j=1}^{k} f_j(\mathbf{w}_j^\top \mathbf{x}),$$

is estimated based on the residuals remaining after the terms estimated so far at each stage have been accommodated. That is, if $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ represent observed pairs of the covariates and response, then initially the mean is estimated as $\hat{\mu} = \bar{y} = \frac{1}{n}\sum_{i=1}^n y_i$ and the residuals are first set to $r_i \leftarrow y_i - \hat{\mu}; i = 1, \ldots, n$. The following is then iterated, for $j = 1, \ldots, k$: (i) estimate $\mathbf{w}_j$ and $f_j(\cdot)$ by minimizing $\sum_{i=1}^n \left(r_i - f_j(\mathbf{w}_j^\top \mathbf{x}_i)\right)^2$; (ii) update the residuals by setting $r_i \leftarrow r_i - f_j(\mathbf{w}_j^\top \mathbf{x}_i); i = 1, \ldots, n$. The main computational and methodological challenge, therefore, is in estimating each of the pairs $\mathbf{w}_j, f_j(\cdot)$, for $j = 1, \ldots, k$. We henceforth drop the subscript j and simply focus on estimating the pair of projection vector, $\mathbf{w}$, and univariate regression function, $f(\cdot)$, from a set of pairs of covariates and residuals, $\{(\mathbf{x}_1, r_1), \ldots, (\mathbf{x}_n, r_n)\}$.

For brevity and simplicity we focus here only on using the Nadaraya-Watson estimator for $f(\cdot)$[1], and for enhanced stability we will use leave-one-out estimates during optimization. The projection index is therefore given by

$$\Phi(\mathbf{w} \mid \mathbf{X}, \mathbf{r}) = \sum_{i=1}^{n} \left(r_i - \frac{\sum_{j \neq i} K\left(\frac{p_j - p_i}{h}\right) r_j}{\sum_{j \neq i} K\left(\frac{p_j - p_i}{h}\right)} \right)^2 \Bigg|_{\mathbf{p} = \mathbf{X}\vec{\mathbf{w}}},$$

where $\mathbf{r}$ is the vector of residuals and $\mathbf{X}$ is the matrix with i-th row given by $\mathbf{x}_i^\top$. It will be convenient going forward to introduce the notation $\boldsymbol{\Sigma}(\cdot)$ and $\boldsymbol{\Sigma}'(\cdot)$ to be the vector valued functions mapping $\mathbb{R}^n \to \mathbb{R}^n$, with

$$\boldsymbol{\Sigma}(\boldsymbol{\omega})_i = \sum_{j \neq i} K\left(\frac{p_j - p_i}{h}\right) \omega_j,$$

[1]The function `fk_ppr()` in **FKSUM** provides implementations of both the Nadaraya-Watson and local linear estimators.

$$\mathbf{\Sigma}'(\boldsymbol{\omega})_i = \sum_{j \neq i} K'\left(\frac{p_j - p_i}{h}\right)\omega_j,$$

where $\mathbf{p} = \mathbf{X}\vec{\mathbf{w}}$, as before, and the value of the projection vector, $\mathbf{w}$, will be clear from the context. For example, we may now write

$$\Phi(\mathbf{w} \mid \mathbf{X}, \mathbf{r}) = \sum_{i=1}^{n}\left(r_i - \frac{\mathbf{\Sigma}(\mathbf{r})_i}{\mathbf{\Sigma}(\mathbf{1})_i}\right)^2 = \sum_{i=1}^{n}(r_i - \hat{r}_i)^2\Big|_{\hat{r}_i = \frac{\mathbf{\Sigma}(\mathbf{r})_i}{\mathbf{\Sigma}(\mathbf{1})_i}},$$

where $\mathbf{1}$ is a vector of ones, and $\hat{\mathbf{r}} = (\hat{r}_1, \dots, \hat{r}_n)$ is the vector of fitted values for the residuals. We are now ready to formulate the objective function to be optimized in R. That is, we define the function `phi_ppr()`, which evaluates the projection index for a given projection vector, as follows

```
R> phi_ppr <- function(w, X, r, h, beta){
+    n <- nrow(X)
+    p <- X %*% w / sqrt(sum(w^2))
+    Sr <- fk_sum(p, r, h, beta = beta) - beta[1] * r
+    S1 <- fk_sum(p, rep(1, n), h, beta = beta) - beta[1]
+    S1[S1 < 1e-20] <- 1e-20
+    r_hat <- Sr / S1
+    sum((r - r_hat)^2)
+  }
```

Here `Sr` and `S1` represent the vectors $\mathbf{\Sigma}(\mathbf{r})$ and $\mathbf{\Sigma}(\mathbf{1})$, respectively. Notice that in order to obtain the leave-one-out terms for the numerator and denominator in the fitted values (`Sr` and `S1` respectively), we need only subtract the vector of coefficients (the argument `omega` in the function), multiplied by `beta[1]`. This is because

$$\mathbf{\Sigma}(\boldsymbol{\omega})_i = \sum_{j \neq i} K\left(\frac{p_j - p_i}{h}\right)\omega_j = \sum_{j=1}^{n} K\left(\frac{p_j - p_i}{h}\right)\omega_j - K(0)\,\omega_i = \sum_{j=1}^{n} K\left(\frac{p_j - p_i}{h}\right)\omega_j - \beta_0\omega_i.$$

Leave-one-out sums can result in values very close to zero if the observations are sparse over some range. When dividing by such terms we find it prudent to buffer them away from zero to avoid numerical instability.

Now, in order to effectively and efficiently optimize this function, we need to evaluate its gradient. In order to compute the gradient, we use the chain rule, as follows,

$$\nabla_{\mathbf{w}}\Phi(\mathbf{w} \mid \mathbf{X}, \mathbf{r})^\top = \nabla_{\mathbf{p}}\left(\sum_{i=1}^{n}\left(r_i - \frac{\sum_{j \neq i} K\left(\frac{p_j - p_i}{h}\right) r_j}{\sum_{j \neq i} K\left(\frac{p_j - p_i}{h}\right)}\right)^2\right)^\top D_{\mathbf{w}}\mathbf{p}\Bigg|_{\mathbf{p} = \mathbf{X}\vec{\mathbf{w}}},$$

where $\nabla_{\mathbf{p}}(\cdot)$ indicates the vector of partial derivatives based on the projected sample, $\mathbf{p} = \mathbf{X}\vec{\mathbf{w}}$, and $D_{\mathbf{w}}\mathbf{p}$ is the matrix whose i, j-th entry is the partial derivative of $p_i = \vec{\mathbf{w}}^\top \mathbf{x}_i$ with respect to w_j, which is equal to $\frac{x_{ij}}{||\mathbf{w}||} - \frac{p_i w_j}{||\mathbf{w}||^2}$. We thus have $D_{\mathbf{w}}\mathbf{p} = \left(\frac{1}{||\mathbf{w}||}\mathbf{X} - \frac{1}{||\mathbf{w}||^2}\mathbf{p}\mathbf{w}^\top\right)$. Now, if we again let $\hat{r}_i = \sum_{j \neq i} K\left(\frac{p_j - p_i}{h}\right) r_j / \sum_{j \neq i} K\left(\frac{p_j - p_i}{h}\right)$, $i = 1, \dots, n$, then the first term in the chain rule product can be determined using

$$
\begin{aligned}
\frac{\partial}{\partial p_i}\sum_{j=1}^n (r_j-\hat r_j)^2 &= \sum_{j=1}^n \frac{\partial}{\partial \hat r_j}(r_j-\hat r_j)^2\,\frac{\partial \hat r_j}{\partial p_i}\\
&= 2\sum_{j=1}^n(\hat r_j - r_j)\,\frac{\partial}{\partial p_i}\frac{\sum_{k\neq j}^n K\left(\frac{p_k-p_j}{h}\right) r_k}{\sum_{k\neq j}^n K\left(\frac{p_k-p_j}{h}\right)}\\
&= 2\sum_{j=1}^n(\hat r_j - r_j)\left(\frac{\sum_{k\neq j}^n y_k\frac{\partial}{\partial p_i}K\left(\frac{p_k-p_j}{h}\right)}{\sum_{k\neq j}^n K\left(\frac{p_k-p_j}{h}\right)} - \frac{\sum_{k\neq j}^n K\left(\frac{p_k-p_j}{h}\right) r_k\sum_{k\neq j}^n\frac{\partial}{\partial p_i}K\left(\frac{p_k-p_j}{h}\right)}{\left(\sum_{k\neq j}^n K\left(\frac{p_k-p_j}{h}\right)\right)^2}\right)\\
&= 2\sum_{j=1}^n\frac{\hat r_j - r_j}{\sum_{k\neq j}^n K\left(\frac{p_k-p_j}{h}\right)}\left(\sum_{k\neq j}^n r_k\frac{\partial}{\partial p_i}K\left(\frac{p_k-p_j}{h}\right) - \hat r_j\sum_{k\neq j}^n\frac{\partial}{\partial p_i}K\left(\frac{p_k-p_j}{h}\right)\right)\\
&= 2\sum_{j\neq i}\frac{\hat r_j - r_j}{\sum_{k\neq j}^n K\left(\frac{p_k-p_j}{h}\right)}\left(\sum_{k\neq j}^n r_k\frac{\partial}{\partial p_i}K\left(\frac{p_k-p_j}{h}\right) - \hat r_j\sum_{k\neq j}^n\frac{\partial}{\partial p_i}K\left(\frac{p_k-p_j}{h}\right)\right)\\
&\quad + 2\frac{\hat r_i - r_i}{\sum_{k\neq i}^n K\left(\frac{p_k-p_i}{h}\right)}\left(\sum_{k\neq i}^n r_k\frac{\partial}{\partial p_i}K\left(\frac{p_k-p_i}{h}\right) - \hat r_i\sum_{k\neq i}^n\frac{\partial}{\partial p_i}K\left(\frac{p_k-p_i}{h}\right)\right)\\
&= 2\sum_{j\neq i}\frac{\hat r_j - r_j}{\sum_{k\neq j}^n K\left(\frac{p_k-p_j}{h}\right)}\left(\frac{r_i}{h}K'\left(\frac{p_i-p_j}{h}\right) - \frac{\hat r_j}{h}K'\left(\frac{p_i-p_j}{h}\right)\right)\\
&\quad - 2\frac{\hat r_i - r_i}{\sum_{k\neq i}^n K\left(\frac{p_k-p_i}{h}\right)}\left(\frac{1}{h}\sum_{k\neq i}^n r_k K'\left(\frac{p_k-p_i}{h}\right) - \frac{\hat r_i}{h}\sum_{k\neq i}^n K'\left(\frac{p_k-p_i}{h}\right)\right)\\
&= \frac{2}{h}\sum_{j\neq i}\frac{\hat r_j(\hat r_j - r_j)}{\sum_{k\neq j} K\left(\frac{p_k-p_j}{h}\right)}K'\left(\frac{p_j-p_i}{h}\right) - \frac{2r_i}{h}\sum_{j\neq i}\frac{(\hat r_j - r_j)}{\sum_{k\neq j} K\left(\frac{p_k-p_j}{h}\right)}K'\left(\frac{p_j-p_i}{h}\right)\\
&\quad + \frac{2\hat r_i(\hat r_i - r_i)}{h\sum_{k\neq i} K\left(\frac{p_k-p_i}{h}\right)}\sum_{j\neq i}K'\left(\frac{p_j-p_i}{h}\right) - \frac{2(\hat r_i - r_i)}{h\sum_{k\neq i} K\left(\frac{p_k-p_i}{h}\right)}\sum_{j\neq i} r_j K'\left(\frac{p_j-p_i}{h}\right),
\end{aligned}
$$

where some of the signs have changed in the final step due to the reversing of the terms inside $K'(\cdot)$, which is an odd function, i.e., $K'(x) = -K'(-x)\ \forall x$. This is now in a form convenient for the notation $\mathbf{\Sigma}(\cdot), \mathbf{\Sigma}'(\cdot)$ used previously. That is, we find

$$
\frac{\partial}{\partial p_i}\sum_{j=1}^n (r_j - \hat r_j) = \frac{2}{h}\left(\mathbf{\Sigma}'\left(\frac{\hat{\mathbf{r}}(\hat{\mathbf{r}}-\mathbf{r})}{\mathbf{\Sigma}(\mathbf{1})}\right)_i - r_i\mathbf{\Sigma}'\left(\frac{\hat{\mathbf{r}}-\mathbf{r}}{\mathbf{\Sigma}(\mathbf{1})}\right)_i + \frac{\hat r_i - r_i}{\mathbf{\Sigma}(\mathbf{1})_i}\left(\hat r_i\mathbf{\Sigma}'(\mathbf{1})_i - \mathbf{\Sigma}'(\mathbf{r})_i\right)\right),
$$

where products and ratios of vectors within $\mathbf{\Sigma}'(\cdot)$ are element-wise operations. To evaluate the gradient, both the kernel and kernel derivative sums of $\mathbf{r}$ and $\mathbf{1}$ are required. We can therefore set `type = "both"` in the function `fk_sum()`. For the other sums, we only require the sums of kernel derivatives. To compute the gradient of the projection index in R, we can thus use the following,

```
R> dphi_ppr <- function(w, X, r, h, beta){
+    n <- nrow(X)
+    nw <- sqrt(sum(w^2))
+    p <- X %*% w / nw
+    S1 <- fk_sum(p, rep(1, n), h, beta = beta, type = "both")
+    S1[, 1] <- S1[, 1] - beta[1]
+    S1[S1[, 1] < 1e-20, 1] <- 1e-20
+    Sr <- fk_sum(p, r, h, beta = beta, type = "both")
+    Sr[, 1] <- Sr[, 1] - beta[1] * r
+    r_hat <- Sr[, 1] / S1[, 1]
+    T1 <- fk_sum(p, r_hat * (r_hat - r) / S1[, 1], h,
+      beta = beta, type = "dksum")
+    T2 <- r * fk_sum(p, (r_hat - r) / S1[, 1], h,
+      beta = beta, type = "dksum")
+    T3 <- (r_hat - r) / S1[, 1] * (r_hat * S1[, 2] - Sr[, 2])
+    dphi_dp <- (T1 - T2 + T3) * 2 / h
+    dp_dw <- (X / nw - p %*% t(w) / nw^2)
+    c(t(dphi_dp) %*% dp_dw)
+  }
```

In the above we have split the partial derivatives with respect to the elements in $\mathbf{p}$ into terms `T1, T2, T3`. We have also used the fact that the function `fk_sum(..., type = "both")` returns a matrix in which the first column contains the kernel sums and the second contains the sums of kernel derivatives. Note that since $K'(0) = 0$ for all symmetric, differentiable kernels, the sums of kernel derivatives are the same whether they are of the leave-one-out type or not. Hence, only the first column in the output of `fk_sum(..., type = "both")` needs to be modified to accommodate the fact that we use leave-one-out sums.

Before continuing, it is prudent to verify that the gradient function is producing the correct output. In order to do so, we compare its output with a numerically approximated gradient based on finite differences. In particular, if a function, $g(\cdot)$, is differentiable, then,

$$\frac{\partial}{\partial w_i} g(\mathbf{w}) = \frac{g(\mathbf{w} + h\mathbf{e}_i) - g(\mathbf{w} - h\mathbf{e}_i)}{2h} + o(h),$$

where $\mathbf{e}_i$ is the vector of zeroes except in the i-th position, where it takes the value one. Note that a simple way to encode the collection of vectors $\{\mathbf{e}_1, \ldots, \mathbf{e}_d\}$ is as the rows of the identity matrix. We begin by generating a set of data (covariates and response variable). We will generate 1000 data points in 10 dimensions from a multivariate Gaussian distribution with a randomly generated covariance matrix. We will then select some random "true" projection vectors, and define the response as a non-linear function of the projected data.

```
R> set.seed(1)
R> n_dat <- 1000
R> n_dim <- 10
```

```
R> X <- matrix(rnorm(n_dat * n_dim), n_dat, n_dim) %*%
+    matrix(2 * runif(n_dim^2) - 1, n_dim, n_dim)
R> wtrue1 <- rnorm(n_dim)
R> wtrue2 <- rnorm(n_dim)
R> y <- (X %*% wtrue1 > 1) * (X %*% wtrue1 - 1) + tanh(X %*% wtrue2 / 2) *
+    (X %*% wtrue1) + (X %*% (wtrue1 - wtrue2) / 5)^2 + rnorm(n_dat)
```

We now check that our exact gradient matches closely to the finite differences approximation, both in a relative and absolute sense, for a randomly selected projection vector. We also, for now, simply choose a random positive bandwidth value.

```
R> w <- rnorm(n_dim)
R> h <- runif(1)
R> beta <- c(0.25, 0.25)
R> Eh <- diag(n_dim) * 1e-5
R> dphi_approx <- apply(Eh, 1, function(eh) (phi_ppr(w + eh, X, y, h, beta)
+    - phi_ppr(w - eh, X, y, h, beta)) / 2e-5)
R> dphi <- dphi_ppr(w, X, y, h, beta)
R> max(abs(dphi / dphi_approx - 1))

[1] 7.004957e-10

R> max(abs(dphi - dphi_approx))

[1] 1.479764e-07
```

We see that the analytical and numerically approximated gradients are extremely close to one another, and so should feel confident that our analytical expression, and the function for evaluating it, is correct.

Next we write a function, `ppr_nw()`, to perform projection pursuit based on minimizing `phi_ppr()`. To optimize the projection we apply the function `optim()`, from R's base **stats** package. Our preferred optimization method is the limited memory BFGS algorithm (Liu and Nocedal 1989). We have found that using an oversmoothing bandwidth during projection pursuit is quite reliable, and tends to be fairly robust as the estimation along each projection has relatively low variation. We select this bandwidth using the optimal rate for univariate regression, $\mathcal{O}(n^{-1/5})$, and a measure of the scale of the covariates, for which we use the square root of the largest eigenvalue of their covariance matrix. Once the optimal projection has been determined, however, the final fitting is performed using a more sophisticated method which is based on leave-one-out cross-validation. It is generally preferable to initialize the projection vector, $\mathbf{w}$, with a sensible heuristic. We use the ordinary least squares solution, with a small ridge to ensure a unique solution, $\mathbf{w} = (\mathbf{X}^\top\mathbf{X} + 0.01\mathbf{I})^{-1}\mathbf{X}^\top\mathbf{y}$. We also allow for alternative initialization with the optional argument `w`, which is by default set to this ridge solution.

```
R> ppr_nw <- function(X, r, w = NULL){
+    n <- nrow(X)
+    d <- ncol(X)
+    if (is.null(w)) w <- solve(t(X) %*% X + 0.01 * diag(d)) %*% t(X) %*% r
```

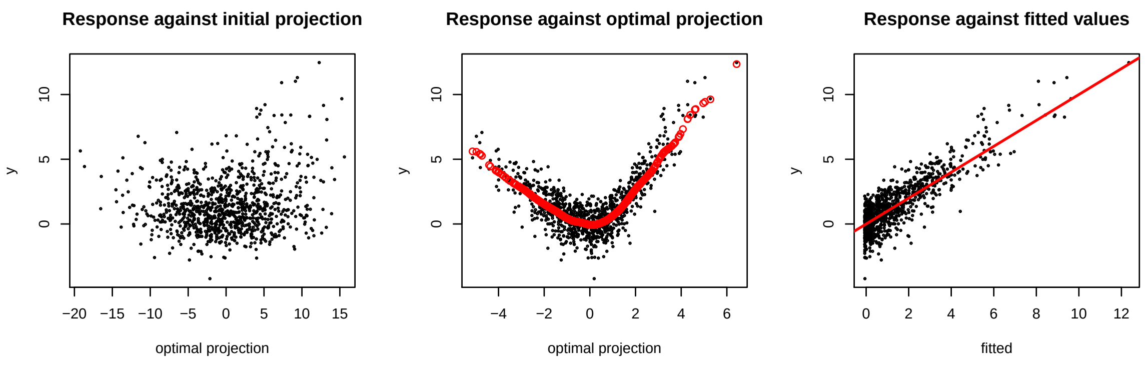


Figure 8: Projection pursuit regression solution from ten dimensional simulated data. The plots show the response against the initial projection, the optimal projection and the fitted values.

```
+     h <- sqrt(eigen(cov(X))$values[1]) / n^0.2
+     w <- optim(w, phi_ppr, dphi_ppr, X, r, h, c(0.25, 0.25),
+       method = "L-BFGS-B")$par
+     w <- w / sqrt(sum(w^2))
+     loo_sse <- function(h) phi_ppr(w, X, r, h, c(0.25, 0.25))
+     h <- optimise(loo_sse, c(h/50, h))$minimum
+     list(w = w, h = h)
+   }
```

We now find the optimal projection for the data which we generated previously, and plot the result. For interest we also plot the response values against the projections along the initial randomly generated projection, as well as against the fitted values. The plots can be seen in Figure 8. Notice that the implementation we have provided above does not necessitate that the mean of the response needs to first be subtracted to obtain the first set of residuals. However, to be consistent with the discussion above, we will include this step.

```
R> mu <- mean(y)
R> r <- y - mu
R> w_opt <- ppr_nw(X, r, w = w)
R> p <- X %*% w_opt$w
R> S1 <- fk_sum(p, rep(1, n_dat), w_opt$h)
R> Sr <- fk_sum(p, r, w_opt$h)
R> fitted <- mu + Sr / S1
R> par(mfrow = c(1, 3))
R> plot(X %*% w, y, main = "Response against initial projection",
+    xlab = "optimal projection", ylab = "y")
R> plot(X %*% w_opt$w, y, main = "Response against optimal projection",
+    xlab = "optimal projection", ylab = "y")
R> points(X %*% w_opt$w, fitted, col = 2)
R> plot(fitted, y, main = "Response against fitted values",
+    xlab = "fitted", ylab = "y")
R> abline(0, 1, lwd = 2, col = 2)
```

3.6. Examples

We will compare the implementation we defined above with the function `ppr()` from R's **stats** package. This function uses an iterative re-weighted least squares approach to optimization, as opposed to a more standard gradient descent. As a result, this implementation is extremely efficient for problems of moderate dimensionality, but scales quadratically in the problem dimension[2].

Simulation

Once again we begin with a simulation to assess the speed and accuracy of our implementation. We consider two scenarios, one of low dimensionality and another of higher dimensionality. Code to reproduce the results of these experiments is available in supplementary material.

In the first experiment we use exactly the same set-up as we did above, with 1000 total observations and 10 covariates. The only difference is that we use half the data for training and the other half to estimate the performance of the models. Here the existing implementation in the **stats** package is superior, with an average running time of 0.003 seconds and average estimated coefficient of determination 0.69, based on 50 replications of the experiment. On the other hand, the implementation we defined above runs an order of magnitude slower and achieves an average coefficient of determination of 0.65.

In the second experiment we consider a higher dimensional example, containing 200 covariates. We also increase the number of data to 5000. In the interest of time, we repeat this experiment only 20 times, and again consider the running time and estimated coefficient of determination. In this case the existing implementation in the **stats** package overfits the data, and obtains a large negative average coefficient of determination, -0.83. On the other hand, our implementation continues to provide accurate predictions with an average coefficient of determination of 0.79. Also, as expected, the running time of the existing implementation increases substantially, now exceeding that of ours by roughly three times.

Baseball salaries

To conclude this section we consider a simple real data example taken from James, Witten, Hastie, and Tibshirani (2013), which is available in the package **ISLR** (James, Witten, Hastie, and Tibshirani 2021). The problem is to predict the salaries of professional baseball players based on various performance statistics. For simplicity, we only use the numerical covariates. We begin by setting up the matrix of covariates and the vector of responses, removing those cases which do not have a response value.

```
R> library("ISLR")
R> X <- as.matrix(Hitters[! is.na(Hitters[, 19]), c(1:13, 16:18)])
R> y <- Hitters[! is.na(Hitters[, 19]), 19]
```

We again compare models based on their estimated coefficient of determination. Because the data set is small, containing only 263 complete observations, we consider multiple splits into 70% training and 30% testing data. We store the training indices in the vector `train`. For interest we also consider models with two components. To estimate the second component, we

[2]Many gradient descent methods scale linearly in the problem dimension.

simply apply the function `ppr_nw()` to the data using the residuals after the first component's fitted values have also been subtracted from the responses.

```
R> n_rep <- 50
R> R2_stats_1 <- numeric(n_rep)
R> R2_stats_2 <- numeric(n_rep)
R> R2_nw_1 <- numeric(n_rep)
R> R2_nw_2 <- numeric(n_rep)
R> for (rep in 1:n_rep) {
+    set.seed(rep)
+    sm <- sample(1:nrow(X), floor(0.7 * nrow(X)))
+    model <- ppr(X[sm,], y[sm], nterms = 1)
+    yhat <- predict(model, X[-sm,])
+    R2_stats_1[rep] <- 1 - mean((yhat - y[-sm])^2) / var(y[-sm])
+    model <- ppr(X[sm,], y[sm], nterms = 2)
+    yhat <- predict(model, X[-sm,])
+    R2_stats_2[rep] <- 1 - mean((yhat - y[-sm])^2) / var(y[-sm])
+    component_1 <- ppr_nw(X[sm,], y[sm])
+    p <- X %*% component_1$w
+    S1 <- fk_sum(p[sm], rep(1, length(sm)), component_1$h, x_eval = p)
+    Sy <- fk_sum(p[sm], y[sm], component_1$h, x_eval = p)
+    fitted <- Sy[sm] / S1[sm]
+    yhat <- Sy[-sm] / S1[-sm]
+    R2_nw_1[rep] <- 1 - mean((yhat - y[-sm])^2) / var(y[-sm])
+    component_2 <- ppr_nw(X[sm,], y[sm] - fitted)
+    p <- X %*% component_2$w
+    S1 <- fk_sum(p[sm], rep(1, length(sm)), component_2$h, x_eval = p[-sm])
+    Sy <- fk_sum(p[sm], y[sm] - fitted, component_2$h, x_eval = p[-sm])
+    yhat <- yhat + Sy / S1
+    R2_nw_2[rep] <- 1 - mean((yhat - y[-sm])^2) / var(y[-sm])
+  }
R> colMeans(cbind(R2_stats_1, R2_stats_2, R2_nw_1, R2_nw_2))

R2_stats_1 R2_stats_2    R2_nw_1    R2_nw_2
 0.3350620  0.2966666  0.3720452  0.4322518
```

Our implementation not only obtains superior accuracy to the existing implementation, but also better utilizes the added flexibility of the additional component, whereas this flexibility appears to lead to overfitting when using the function `ppr()` in this instance.

4. Conclusion

In this paper we discussed the use of the R package **FKSUM** for efficient univariate kernel smoothing and projection pursuit when the projection index requires evaluating kernel based estimates of functionals of the projected data distribution. These include independent component analysis; projection pursuit regression and minimum density hyperplanes.

In all cases we investigated the accuracy and efficiency of the package's implementation in comparison with existing implementations available either with R's standard distribution or through the comprehensive R archive network. In all cases the implementations provided in **FKSUM** showed very competitive performance. In addition we provided a detailed derivation and implementation of projection pursuit regression, which we believe should provide readers with sufficient instruction to use the package functionality for implementing smoothing and projection pursuit methods in R which lie beyond the scope of the package in its current form.

References

Amari S, Cichocki A, Yang HH (1996). "A New Learning Algorithm for Blind Signal Separation." In *Advances in Neural Information Processing Systems*, pp. 757–763.

Bach FR, Jordan MI (2002). "Kernel Independent Component Analysis." *Journal of Machine Learning Research*, **3**(Jul), 1–48. doi:10.1109/icassp.2003.1202783.

Bolton RJ, Krzanowski WJ (2003). "Projection Pursuit Clustering for Exploratory Data Analysis." *Journal of Computational and Graphical Statistics*, **12**(1), 121–142. doi:10.1198/1061860031374.

Bowman A, Azzalini A (2021). ***sm**: Smoothing Methods for Nonparametric Regression and Density Estimation.* R package version 2.2-5.7, URL https://CRAN.R-project.org/package=sm.

Chen A (2006). "Fast Kernel Density Independent Component Analysis." In *International Conference on Independent Component Analysis and Signal Separation*, pp. 24–31. Springer-Verlag. doi:10.1007/11679363_4.

Dua D, Karra Taniskidou E (2017). "UCI Machine Learning Repository." URL http://archive.ics.uci.edu/ml/.

Duong T (2021). ***ks**: Kernel Smoothing.* R package version 1.13.2, URL https://CRAN.R-project.org/package=ks.

Eddelbuettel D, François R (2011). "**Rcpp**: Seamless R and C++ Integration." *Journal of Statistical Software*, **40**(8), 1–18. doi:10.18637/jss.v040.i08.

Fan J, Marron JS (1994). "Fast Implementations of Nonparametric Curve Estimators." *Journal of Computational and Graphical Statistics*, **3**(1), 35–56. doi:10.1080/10618600.1994.10474629.

Friedman JH (1987). "Exploratory Projection Pursuit." *Journal of the American Statistical Association*, **82**(397), 249–266. doi:10.1080/01621459.1987.10478427.

Friedman JH, Stuetzle W (1981). "Projection Pursuit Regression." *Journal of the American Statistical Association*, **76**(376), 817–823. doi:10.1080/01621459.1981.10477729.

Hall P, Wand MP (1996). "On the Accuracy of Binned Kernel Density Estimators." *Journal of Multivariate Analysis*, **56**(2), 165–184. doi:10.1006/jmva.1996.0009.

Hastie T, Tibshirani R (2003). "Independent Components Analysis Through Product Density Estimation." In *Advances in Neural Information Processing Systems*, pp. 665–672.

Hastie T, Tibshirani R (2010). ***ProDenICA**: Product Density Estimation for ICA Using Tilted Gaussian Density Estimates.* R package version 1.0, URL `https://CRAN.R-project.org/package=ProDenICA`.

Hofmeyr D, Pavlidis N (2015). "Maximum Clusterability Divisive Clustering." In *2015 IEEE Symposium Series on Computational Intelligence*, pp. 780–786. `doi:10.1109/ssci.2015.116`.

Hofmeyr DP (2017). "Clustering by Minimum Cut Hyperplanes." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **39**(8), 1547–1560. `doi:10.1109/tpami.2016.2609929`.

Hofmeyr DP (2021). "Fast Exact Evaluation of Univariate Kernel Sums." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, **43**, 447–458. `doi:10.1109/tpami.2019.2930501`.

Hofmeyr DP (2022). ***FKSUM**: Fast Kernel Sums.* R package version 1.0.0, URL `https://CRAN.R-project.org/package=FKSUM`.

Hofmeyr DP, Pavlidis NG (2019). "**PPCI**: An R Package for Cluster Identification Using Projection Pursuit." *The R Journal*, **11**(2), 152–170. `doi:10.32614/rj-2019-046`.

Huber PJ (1985). "Projection Pursuit." *The Annals of Statistics*, **13**(2), 435–475. `doi:10.1214/aos/1176349519`.

Hyvärinen A, Oja E (2000). "Independent Component Analysis: Algorithms and Applications." *Neural Networks*, **13**(4), 411–430. `doi:10.1016/s0893-6080(00)00026-5`.

James G, Witten D, Hastie T, Tibshirani R (2013). *An Introduction to Statistical Learning*, volume 112. Springer-Verlag.

James G, Witten D, Hastie T, Tibshirani R (2021). ***ISLR**: Data for an Introduction to Statistical Learning with Applications in R.* R package version 1.4, URL `https://CRAN.R-project.org/package=ISLR`.

Karvanen J, Koivunen V (2002). "Blind Separation Methods Based on Pearson System and Its Extensions." *Signal Processing*, **82**(4), 663–673. `doi:10.1016/s0165-1684(01)00213-4`.

Kullback S, Leibler RA (1951). "On Information and Sufficiency." *The Annals of Mathematical Statistics*, **22**(1), 79–86. `doi:10.1214/aoms/1177729694`.

Langrené N, Warin X (2019). "Fast and Stable Multivariate Kernel Density Estimation by Fast Sum Updating." *Journal of Computational and Graphical Statistics*, **28**(3), 596–608. `doi:10.1080/10618600.2018.1549052`.

Liu DC, Nocedal J (1989). "On the Limited Memory BFGS Method for Large Scale Optimization." *Mathematical Programming*, **45**(1-3), 503–528. `doi:10.1007/bf01589116`.

Marchini JL, Heaton C, Ripley BD (2021). ***fastICA**: FastICA Algorithms to Perform ICA and Projection Pursuit.* R package version 1.2-3, URL https://CRAN.R-project.org/package=fastICA.

Miettinen J, Nordhausen K, Taskinen S (2017). "Blind Source Separation Based on Joint Diagonalization in R: The Packages **JADE** and **BSSasymp**." *Journal of Statistical Software*, **76**(2), 1–31. doi:10.18637/jss.v076.i02.

Morrissey M, Sakrejda K (2014). ***gsg**: Calculation of Selection Coefficients.* R package version 2.0, URL https://CRAN.R-project.org/src/contrib/Archive/gsg/.

Nadaraya EA (1964). "On Estimating Regression." *Theory of Probability & Its Applications*, **9**(1), 141–142. doi:10.1137/1109020.

Niu D, Dy JG, Jordan MI (2011). "Dimensionality Reduction for Spectral Clustering." In *International Conference on Artificial Intelligence and Statistics*, pp. 552–560.

Ossani PC, Cirillo MA (2021). ***Pursuit**: Projection Pursuit.* R package version 1.0.2, URL https://CRAN.R-project.org/package=Pursuit.

Pavlidis NG, Hofmeyr DP, Tasoulis SK (2016). "Minimum Density Hyperplanes." *Journal of Machine Learning Research*, **17**(156), 1–33.

R Core Team (2021). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Scott DW, Sheather SJ (1985). "Kernel Density Estimation with Binned Data." *Communications in Statistics – Theory and Methods*, **14**(6), 1353–1359. doi:10.1080/03610928508828980.

Silverman BW (1982). "Algorithm AS 176: Kernel Density Estimation Using the Fast Fourier Transform." *Journal of the Royal Statistical Society C*, **31**(1), 93–99. doi:10.2307/2347084.

Silverman BW (1986). *Density Estimation for Statistics and Data Analysis.* Routledge, Boca Raton. doi:10.1201/9781315140919.

Stroustrup B (2013). *The C++ Programming Language.* 4th edition. Addison-Wesley.

Thrun MC, Ultsch A (2020). "Using Projection Based Clustering to Find Distance and Density Based Clusters in High-Dimensional Data." *Journal of Classification*, **38**, 280–312. doi:10.1007/s00357-020-09373-2.

Wand MP (2021). ***KernSmooth**: Functions for Kernel Smoothing Supporting Wand & Jones (1995).* R package version 2.23-20, URL https://CRAN.R-project.org/package=KernSmooth.

Watson GS (1964). "Smooth Regression Analysis." *Sankhya A*, **26**(4), 359–372.

Yang C, Duraiswami R, Gumerov NA, Davis L (2003). "Improved Fast Gauss Transform and Efficient Kernel Density Estimation." In *Proceedings of Ninth IEEE International Conference on Computer Vision*, pp. 664–671.

Affiliation:

David P. Hofmeyr
Department of Statistics and Actuarial Science
Faculty of Economics and Management Science
Stellenbosch University
Stellenbosch, 7600, South Africa
E-mail: dhofmeyr@sun.ac.za

Journal of Statistical Software https://www.jstatsoft.org/
published by the Foundation for Open Access Statistics https://www.foastat.org/

January 2022, Volume 101, Issue 3 *Submitted:* 2020-02-14
doi:10.18637/jss.v101.i03 *Accepted:* 2021-02-02