Appendix

A.1. Implementing DP Mixtures in R

We introduce some R macros to implement inference in a DP mixture in (3.9). Using Gaussian kernels the DP mixture model becomes

$$y_i \mid G \sim F(y_i) = \int N(y_i;\ \theta_i, \sigma)\ dp(G). \tag{A.10}$$

We will use $f(x)$ to denote the p.d.f. The moel can equivalently be written as a hierarchical model

$$y_i \mid \theta_i \sim N(\theta_i, \sigma), \qquad \theta_i \mid G \sim G. \tag{A.11}$$

where $G \sim \mathsf{DP}(M, G_0)$. For the centering measure, we use $G_0 = N(m_0, B_0)$, and we take $m_0 = 0$ and $B_0 = 4$, while the precision parameter is set to $M = 1$. We complete the model with a gamma prior for the kernel width, $1/\sigma^2 \sim \mathsf{Ga}(a, b)$; in the example below we take $a = b = 1$. We implement posterior MCMC simulation using the methods described in §3.3. The complete R code is available at

`http://www.math.utexas.edu/users/pmueller/prog/BNPnotes/`

We briefly explain the main steps in the macros. We use the data from the Old Faithful geyser data in R and fix the hyperparameters for the model. The hyperparameters and the data are saved as global variables:

```
## DATA: Old Faithful geyser data
y <- round(faithful$eruptions, digits=2)
n <- length(y)
## hyperparameters
a  <- 1;   b <- 1    # 1/sig ~ Ga(a,b)
m0 <- 0;  B0 <- 4    # G0 = N(m0,B0)
M  <- 1
```

Collapsed Gibbs Sampler – Conjugate Models

We first implement the Gibbs sampler from §3.3.1. The algorithm consists of the following steps:

0. Initialization: We initialize s_i using (deterministic) hierarchical clustering.
Initialize a plot by plotting a kernel density estimate of $f(y)$.
1. Update θ_i: sample from $p(\theta_i \mid \boldsymbol{\theta}_{-i}, \sigma, \mathbf{y})$, $i = 1, \ldots, n$. See (3.13).
2. Update $\theta_j^\star$: sample from $p(\theta_j^\star \mid s_i, \sigma, \mathbf{y})$, $j = 1, \ldots, k$, as in (3.14).
3. Update σ^2: sample from $p(\sigma^2 \mid \mathbf{s}, \boldsymbol{\theta}^\star, \mathbf{y})$.
4. Generate f: generate $f \sim p(f \mid \boldsymbol{\theta}, \phi, \sigma)$; add f to the plot.

In Step 4 we sample f conditional on the currently imputed values of $\boldsymbol{\theta}, \phi, \sigma$. For plotting it suffices to evaluate f on a grid that is fine enough to get a smooth plot. To sample f (on the grid) we use a minor approximation. Essentially we (i) sample $G \sim p(G \mid \boldsymbol{\theta})$, and (ii) evaluate $F = \int \mathsf{N}(\theta, \sigma)\, dG(\theta)$. The approximation

is applied in step (i); recall that $p(G \mid \theta_1, \ldots, \theta_n) = \mathsf{DP}(M+n, G_1)$ with $G_1 \propto MG_0 + \sum_{i=1}^{n} \delta_{\theta_i}$. For large n, the total mass (precision) $M_1 = M+n$ is large. In the limiting case $(M+n) \to \infty$ little uncertainty is left, and $G \approx G_1$. The code below uses the limiting case as an approximation, hence, we (approximately) generate $F \sim p(F \mid \boldsymbol{\theta}, \sigma)$ as

$$F \propto \sum n_j p(y \mid \theta_j^\star) + M \int p(y \mid \theta) dp G^\star(\theta).$$

This is a poor man's version of the algorithm proposed by Gelfand and Kottas (2002).

Steps 0 through 4 are implemented in the following R macros. We first define macros for each of the transition probabilities and for sampling $f \sim p(f \mid \boldsymbol{\theta})$. A final macro `gibbs()` implements the loop across iterations and calls each of the transition probabilities in turns.

Step 0. As with any MCMC algorithm, a good initialization can significantly speed up convergence. Good initializations might use (for example) exploratory data analysis estimates, empirical Bayes estimates or maximum likelihood estimates. In this case we use an initialization with a partition that is created by cutting a (deterministic) hierarchical clustering tree, for say $k = 10$ clusters. The macro `init.DPk()` implements this initialization. Recall that the data `y` is available as a global variable.

```
init.DPk <- function()
  { ## inital EDA estimate of th[1..n]

    ## cluster data, and cut at height H=10, to get 10 clusters
    hc <- hclust(dist(y)^2, "cen")
    s  <- cutree(hc, k = 10)          # cluster membership indicators
    ths <- sapply(split(y,s),mean)    # cluster specific means
    th <-  ths[s]                     # return th_i = ths[ s_i ]
    return(th)
}
```

Step 1. The first transition probability generates from $p(\theta_i \mid \boldsymbol{\theta}_{-i}, \sigma, \mathbf{y})$, $i = 1, \ldots, n$.

```
sample.th <- function(th,sig)
  { ## sample
    ##    th[i]  ~ p(th_i | th[-i],sig,y)
    ## returns updated th vector

    for(i in 1:n){
      ## unique values and counts
      nj <- table(th[-i])             # counts
      ths <- as.numeric(names(nj))    # unique values
      k <- length(nj)
      ## likelihood
      fj <- dnorm(y[i], m=ths, s=sig)           # p(y_i | th*_j, sig), j=1..k
      f0 <- dnorm(y[i], m=0,   s=sqrt(4+sig^2)) # q0
      pj <- c(fj*nj,f0*M)                       # p(s[i]=j | ...), j=1..k, k+1
      s <- sample(1:(k+1), 1, prob=pj)          # sample s[i]
      if (s==k+1){ ## generate new th[i] value
        v.new <- 1.0/(1/B0 + 1/sig^2)
        m.new <- v.new*(1/B0*m0 + 1/sig^2*y[i])
        thi.new   <- rnorm(1,m=m.new,sd=sqrt(v.new))
        ths <- c(ths,thi.new)
      }
```

```
    th[i] <- ths[s]                              # record new th[i]
  }
  return(th)
}
```

Step 2. Next we update $\theta_j^\star$.

```
sample.ths <- function(th,sig)
  { ## sample ths[j] ~ p(ths[j] | ....)
    ##               = N(ths[j]; m0,B0) * N(ybar[j]; ths[j], sig2/n[j])
    ##               = N(ths[j]; mj,vj)

    ## unique values and counts
    nj <- table(th)                 # counts
    ths <- sort(unique(th))         # unique values
           ##use sort(.) to match table counts
    k <- length(nj)
    for(j in 1:k){
      ## find Sj={i: s[i]=j} and compute sample average over Sj
      idx <- which(th==ths[j])
      ybarj <- mean(y[idx])
      ## posterior moments for p(ths[j] | ...)
      vj <- 1.0/(1/B0 + nj[j]/sig^2)
      mj <- vj*(1/B0*m0 + nj[j]/sig^2*ybarj)
      thsj <- rnorm(1,m=mj,sd=sqrt(vj))
      ## record the new ths[j] by replacing all th[i], i in Sj.
      th[idx] <- thsj
    }
    return(th)
  }
```

Step 3. The last transition probability in each iteration updates σ^2 using $p(1/\sigma^2 \mid \boldsymbol{\theta}, \boldsymbol{y}) \propto \mathsf{Ga}(a_1, b_1)$ with $a_1 = a + 0.5n$ and $b_1 = b + 0.5s^2$, where $s^2 = \sum_i (y_i - \theta_i)^2$ is the residual sum of squares.

```
sample.sig <- function(th)
  { ## sample
    ##   sig ~ p(sig | ...)
    ## returns: sig

    s2 <- sum( (y-th)^2 )    # sum of squared residuals
    a1 <- a+0.5*n
    b1 <- b+0.5*s2
    s2.inv <- rgamma(1,shape=a1,rate=b1)
    return(1/sqrt(s2.inv))
  }
```

Step 4. The macro `fbar()` implements the draw from $f \sim p(f \mid \theta^\star, \sigma)$.

```
fbar <- function(x,th,sig)
  { ## conditional draw F ~ p(F | th,sig,y) (approx -- will talk about this..)
    ##

    nj <- table(th)              # counts
    ths <- as.numeric(names(nj)) # unique values
    k <- length(nj)
    fx <- M/(n+M)*dnorm(xgrid,m=m0,sd=sqrt(B0+sig))
    for(j in 1:k)
      fx <- fx + nj[j]/(n+M)*dnorm(xgrid,m=ths[j],sd=sig)
    return(fx)
  }
```

Gibbs loop: The macro `gibbs()` implements `n.iter` steps of the MCMC, calling

in turn macros for each of the transition probabilities. Before the actual for loop, the macro initializes several lists that will accumulate the states in each iteration, and starts a plot of a simple kernel density estimate, to which the draws $f \sim p(f \mid \boldsymbol{y})$ will be added.

```
gibbs <- function(n.iter=100)
  {
    th <- init.DPk()              ## initialize th[1..n]
    sig <- sqrt( mean((y-th)^2))  ## and sig
    ## set up data structures to record imputed posterior draws..1
    xgrid <- seq(from=0,to=6,length=50)
    fgrid <- NULL     ## we will record imputed draws of f
    njlist <- NULL    ## record sizes of 8 largest clusters
    klist <- NULL
    ## start with a plot of a kernel density estimate of the data
    plot(density(y),xlab="X",ylab="Y",bty="l",type="l",
         xlim=c(0,6),ylim=c(0,0.7), main="")
    ## now the Gibbs sampler
    for(iter in 1:n.iter){
      th  <- sample.th(th,sig)    ## 1. [th_i  | ...]
      sig <- sample.sig(th)       ## 2. [sig   | .. ]
      th  <- sample.ths(th,sig)   ## 3. [ths_j | ...]
      ## update running summaries ##################
      f   <- fbar(xgrid,th,sig)
      lines(xgrid,f,col=iter,lty=3)
      fgrid <- rbind(fgrid,f)
      nj <- table(th)                 # counts
      njlist <- rbind(njlist,sort(nj,decr=T)[1:8])
      klist <- c(klist,length(nj))
    }
    ## report summaries ############################
    fbar <- apply(fgrid,2,mean)
    lines(xgrid,fbar,lwd=3,col=2)
    njbar <- apply(njlist,2,mean,na.rm=T)
    cat("Average cluster sizes:\n",format(njbar),"\n")
    pk <- table(klist)/length(klist)
    cat("Posterior probs p(k): (row1 = k, row2 = p(k) \n ")
    print(pk/sum(pk))
    return(list(fgrid=fgrid,klist=klist,njlist=njlist))
  }
```

The defined macros could be called as follows:

```
mcmc <- gibbs()
names(mcmc)

## report summaries
njbar <- apply(mcmc$njlist,2,mean,na.rm=T)
cat("Average cluster sizes:\n",format(njbar,digits=1),"\n")
pk <- table(mcmc$klist)/length(mcmc$klist)
cat("Posterior probs p(k): (row1 = k, row2 = p(k) \n ")
print(pk/sum(pk))
```

A blocked Gibbs sampler for the finite DP

Alternatively we implement the Gibbs sampler for a DP mixture of normals (A.10) as before, but now with a finite DP prior, $G \sim \mathsf{DP}_H(M, G_0)$. We implement Gibbs sampling posterior simulation using (3.23) through (3.25). We need an additional hyperparameter to fix H in the finite DP. The initialization proceeds in the same

way as before using a hierarchical clustering. However, now we use the solution from the hierarchical clustering tree to initialize the $\widetilde{\theta}_h$ and w_h parameters.

```
H <- 10    # additional hyperpar

init.DPk <- function()
  { ## inital EDA estimate of G = sum_{h=1..10} w_h delta(m_h)
    ## returns:
    ##   list(mh,wh)
    ## use (mh,wh) to initialize the blocked Gibbs

    ## cluster data, and cut at height H=10, to get 10 clusters
    hc <- hclust(dist(y)^2, "cen")
    r  <- cutree(hc, k = 10)
    ## record cluster specific means, order them
    mh1 <- sapply(split(y,r),mean)    # cluster specific means == m_h
    wh1 <- table(r)/n
    idx <- order(wh1,decreasing=T)    # re-arrange in deceasing order
    mh <- mh1[idx]
    wh <- wh1[idx]
    return(list(mh=mh,wh=wh))
}
```

The next three R macros implement sampling from distributions (3.23) through (3.25). A fourth transition probability to update σ^2 remains unchanged from before, and we continue to use the earlier defined macro `sample.sig()`. A final macro `gibbs.H()` implements the loop over iterations.

```
sample.r <- function(wh,mh,sig)
  { ## samle allocation indicators

    r <- rep(0,n)
    for(i in 1:n){
      ph <-   dnorm(y[i],m=mh,sd=sig)*wh # likelihood   * prior
                                         ## p(yi | ri=h) * w_h
      r[i] <- sample(1:H,1,prob=ph)
    }
    return(r)
  }


sample.mh <- function(wh,r)
  { ## sample mh ~ p(mh | ...)
    ##

    mh <- rep(0,H)     # initialize
    for(h in 1:H){
      if(any(r==h)){      # some data assigned to h-th pointmass
        Sh <- which(r==h) # Sh = {i: r[i]=h
        nh <- length(Sh)
        ybarh <- mean(y[Sh])
        varh   <- 1.0/(1/B0 + nh/sig^2)
        meanh  <- varh*(1/B0*m0 + nh/sig^2*ybarh)
      } else {            # no data assinged to h-th pointmass
        varh  <- B0       # sample from base measure
        meanh <- m0
      }
      mh[h] <- rnorm(1,m=meanh,sd=sqrt(varh))
    }
    return(mh)
  }
```

```
sample.vh <- function(r)
  {## sample vh ~ p(vh | ...)
   ## returns: wh

    vh <- rep(0,H)  # initialize
    wh <- rep(0,H)
    V <-  1         # record prod_{g<h} (1-vh_h)
    for(h in 1:(H-1)){
      Ah <- which(r==h)
      Bh <- which(r>h)
      vh[h] <-  rbeta(1, 1+length(Ah), M+length(Bh))
      wh[h] <- vh[h]*V
      V <- V*(1-vh[h])
    }
    vh[H] <- 1.0
    wh[H] <- V
    return(wh)
  }
```

Imputing the density f is much simpler now. We can literally evaluate the mixture of normals.

```
fbar.H <- function(xgrid,wh,mh,sig)
  { \#\# return a draw F ~ p(F | ...) (approx)

    fx <- rep(0,length(xgrid))
    for(h in 1:H)
      fx <- fx + wh[h]*dnorm(xgrid,m=mh[h],sd=sig)
    return(fx)
  }
```

An outer loop over iterations implements the Gibbs sampler. The macro looks very similar to before. The Gibbs sampler is executed by calling `fgrid <- gibbs.H()`.

```
gibbs.H <- function(n.iter=100)
  {

    DPk <- init.DPk()
    sig <- 0.11
    wh <- DPk$wh
    mh <- DPk$mh

    ## data structures to save imputed F ~ p(F | ...)
    xgrid <- seq(from=0,to=6,length=50)
    fgrid <- NULL
    plot(density(y),xlab="X",ylab="Y",bty="l",type="l",
         xlim=c(0,6),ylim=c(0,0.7), main="")
    ## Gibbs
    for(iter in 1:n.iter){
      r <-  sample.r(wh,mh,sig)   # 1. r_i ~ p(r_i | ...), i=1..n
      mh <- sample.mh(wh,r)       # 2. m_h ~ p(m_h | ...), h=1..H
      vh <- sample.vh(r)          # 3. v_h ~ p(v_h | ...), h=1..H
      th <- mh[r]                 # record implied th[i] = mh[r[i]]
      sig <- sample.sig(th)       # 4. sig ~ p(sig | ...)

      ## record draw F ~ p(F | th,sig,y) (approx)
      f   <- fbar.H(xgrid,wh,mh,sig)
      lines(xgrid,f,col=iter,lty=3)
      fgrid <- rbind(fgrid,f)
    }
    ## add overall average (= posterior mean) to the plot
    fbar <- apply(fgrid,2,mean)
```

```
    lines(xgrid,fbar,lwd=3,col=2)
    return(fgrid)
  }
```

DPpackage

The detailed R code is useful to understand the algorithm. However, for actual data analysis the use of implementations in available public domain R packages is preferable. For example, `DPpackage` implements MCMC for DP mixtures. The following R fragment shows the use of `DPpackage` with the same Old Faithful geyser data. For more detail see Jara *et al.* (2011).

```
require ("DPpackage")

##### set up parameters for call to DPdensity(.) below:
state <- NULL                                          ## Initial state
nburn<-10; nsave<-1000; nskip<-10; ndisplay<-100       ## MCMC parameters
mcmc <- list(nburn=nburn,nsave=nsave,nskip=nskip,ndisplay=ndisplay)

## Prior 1: fixed alpha, m1, and Psi1
prior1<-list(alpha=1,m1=rep(0,1),psiinv1=diag(0.5,1),nu1=4,tau1=1,tau2=100)

## Prior 4: everything is random
prior4<-list(a0=2,b0=1,m2=rep(0,1),s2=diag(100000,1),
             psiinv2=solve(diag(0.5,1)),
             nu1=4,nu2=4,tau1=1,tau2=100)

## Fit the models and plot the density estimates
fit1 <-DPdensity(y=y,prior=prior1,mcmc=mcmc,state=state,status=TRUE)
fit4 <-DPdensity(y=y,prior=prior4,mcmc=mcmc,state=state,status=TRUE)
plot(fit1,ask=FALSE)
plot(fit4,ask=FALSE)
```

A.2. Implementing PTs in R

One of the attractions of using PT priors is the straightforward implementation of posterior inference. We show some R macros that implement inference in a density estimation problem

$$x_i \mid G \sim G, \qquad\qquad G \sim \mathsf{PT}(\mathcal{A}, G_0).$$

The nested sequence of partitions is defined by the quantiles of a centering measure G_0 (recall the discussion in §4.1). We use a Gaussian centering distribution $G_0(x) = N(\mu, \sigma)$ with fixed hyperparameters $\mu = 0$ and $\sigma = 1$. We implement inference for a Pólya tree up to $M = 5$ levels of nested partitions.

We first generate simulated data from a log normal distribution with some outliers and fixed hyperparameters. After generating the data we produce a histogram.

```
M  <- 5                ## number of levels in PT prior
mu <- 0; sig <- 1      ## centering measure G0 = N(mu,sig)
alpha <- 1
y <- make.dta()

make.dta <- function(plt=T)
  { ## prepares data
    ##
    w <- rnorm(134,m= -1, s=0.5)
    x <-  exp(w)
    x <-  c(x,3.866, 189.3)
    y <- log(x)
    n <-  length(y)
    if (plt)
      hist(log(x),nclass=15)
    return(y)
  }
```

R macros for posterior simulation and means.

We record the boundaries of the partitioning subsets B_ϵ, defined by dyadic quantiles of G_0. This is done in the R macro `prior.pars()`.

```
## record nested partition sequence B[eps]
prior.pars <- function()
  { ## sets up (right) boundaries of partitioning subsets for
    ## PT centered at N(mu,sig)
    B <- as.list(1:M)
    for(m in 1:M){
      q <- 1/(2**m)*(1:(2**m-1))     ## part sets (right boundaries of sets)
      B[[m]] <- c(qnorm(q,m=mu,sd=sig),99)
    }
    return(B)
  }
```

Another macro updates the prior parameters We use prior parameters $\alpha_\epsilon = \alpha m^2$ and compute posterior parameters as in (4.1).

```
## update PT parameters
post.pars <- function(B,alpha,y)
  { ## posterior pars for a PT prior with
    ## PT(B, A) with (right) boundaries of the partitioning sequence B
    ##          defined in B[[1]]..B[[M]] for levels 1..M
    ##          alpha[ep1...epsm] = alpha*m^2
```

```
  M <- length(B)
  neps <- list(1:M)
  a <- list(1:M)
  for(m in 1:M){
    neps[[m]] <- 0*(1:2**m)              ## initialize counts
    a[[m]]    <- alpha*m**2*rep(1,2**m)  ## prior beta parameters
    for(i in 1:n){
      j <- which(B[[m]]>y[i])[1]
      neps[[m]][j] <-  neps[[m]][j]+1    ## update the counts
    }
    a[[m]] <- a[[m]]+neps[[m]]           ## updated beta pars
  }
  return(a)
}
```

Now all parameters are set to proceed with posterior simulation. We define one macro, `post.sim()`, to carry out both, posterior simulation of $G \sim p(G \mid \overline{x})$ and the evaluation of $\overline{G} = E(G \mid \overline{x})$. An argument `sim` $\in \{T, F\}$ selects posterior simulation (T) or expectation (F). In the latter case the simulation from the beta random variables for the conditional splitting probabilities Y_ϵ are replaced by their expectations. Since all splitting probabilities are independent, by definition of the PT, the expectation of the product is the product of the expectations.

```
## posterior draw and posterior mean prob's
post.sim <- function(B,a,sim=T)
  { ## posterior simulation (sim=T) or mean (sim=F)
    M <- length(B)

    Y <- as.list(1:M)    ##  random splitting probs Y[eps]     (sim=T)
                         ##  mean splitting prob E(Y[eps] | y) (sim=F)
    P <- as.list(1:M)    # (random) prob's G(B[eps]) of partitioning subsets
                         ## when (sim=F):
                         ## Y = E(Y[eps] | y) and P = E{ G(B[eps]) | y}
    ## loop over all levels, m=0..M-1
    ## generate random splitting prob's starting with level 1,
    ## i.e. splitting sample space from level m=0
    ## split at level m=M-1 creates lowest level sets at m=M
     for(m in 0:(M-1)){
      for(j in 1:2**m){    # note, for m=0, only j=1 for the sample space
        j0 <- (j-1)*2 + 1  # index of left descendant set B[eps0]
        j1 <- (j-1)*2 + 2  # ... right set B[eps1]
        a0 <- a[[m+1]][j0]
        a1 <- a[[m+1]][j1]
        Y0 <- ifelse(sim, rbeta(1,a0,a1), a0/(a0+a1))
              ## (sim=T): generate random Beta splitting prob
              ## (sim=F): record mean splitting prob E(Y[eps] | dta)
        Y[[m+1]][j0] <- Y0
        Y[[m+1]][j1] <- 1-Y0
        if (m>0){
          P[[m+1]][j0] <- Y[[m+1]][j0] * P[[m]][j]
          P[[m+1]][j1] <- Y[[m+1]][j1] * P[[m]][j]
        } else {
          P[[m+1]][j0] <- Y[[m+1]][j0]
          P[[m+1]][j1] <- Y[[m+1]][j1]
        }
      }
    }
    return(list(Y=Y,P=P))
  }
```

The function `plt.G()` plots a simulated distribution G or a posterior mean $\overline{G}$ when

the distribution is given by probabilities over the partitioning subsets.

```
## plotting a distribution with probabilities P[eps]=p(B[eps])
plt.G <- function(P,B,col=1,lwd=3,lty=1)
    ## We build up the density P as a line (xx,yy)
    xx <- NULL
    yy <- NULL
    for(j in 1:2**M){ # loop over all part subsets B[eps] at level M
      ## we record endpoints: B[eps] = [x0,x1]
      if (j==1)       # left endpoint
        x0 <- -3
      else
        x0 <- B[[M]][j-1]
      if (j==2**M)    # right endpoint
        x1 <- 3
      else
        x1 <- B[[M]][j]
      ## and the density value = probability / length of interval
      y01 <- P[[M]][j]/(x1-x0)
      ## and now add the segment over B[eps] to (xx,yy)
      xx <- c(xx,x0,x1)
      yy <- c(yy,y01,y01)
    }
    ## finally, plot it..
    lines(xx,yy,lwd=lwd,col=col,lty=lty)
  }
```

The defined macros can be executed, for example, by the following sequence of calls.

```
    y  <- make.dta()            # prepare data set
    B <- prior.pars()           # partition boundaries
    a <- post.pars(B,alpha,y)   # posterior parameters

    ## plot the data
    hist(y,breaks=15,prob=T,ylim=c(0,0.9),main="")

    ## 20 random posterior draws
    for(i in 1:20){
      G <- post.sim(B,a)
      plt.G(G$P,B,col=1,lwd=1,lty=2)
    }
    ## posterior mean E(G | y)
    Gbar <- post.sim(B,a,sim=F)
    plt.G(Gbar$P,B,col=2,lwd=3)
```

DPpackage

The detailed R code is useful to understand the algorithm. For actual data analysis the use of implementations in available public domain R packages is much preferable. For example `DPpackage` implements MCMC for PT priors. In this case we use a PT mixture, as in §4.4. The following R fragment shows the use of `DPpackage` with the same data set. For more detail see Jara *et al.* (2011).

```
y  <- make.dta()                                ## prepare data set
state <- NULL                                   ## MCMC parameters
mcmc <- list(nburn=1000,nsave=1000,nskip=50,ndisplay=100,
             tune1=0.15,tune2=1.1,tune3=1.1)
prior<-list(alpha=1,M=6)                        ## Prior information
                                                ## Fitting the model
```

```
fit1 <- PTdensity(y=y,ngrid=1500,prior=prior,mcmc=mcmc,
                  state=state,status=TRUE)
hist(y,breaks=15,prob=T,ylim=c(0,0.8),main="") ## plot the data
lines(fit1$x1,fit1$dens,lty=1,lwd=3)           ## add PT fit
lines(density(y),col=2,type="l")               ## kernel density fit
```

Using the R package we can make use of many built-in additional features. For example, the package includes the evaluation of pseudo marginal likelihood values that can be used for model comparison.