Parallelizing MCMC with Random Partition Trees

-

Application to Book Crossing Dataset

Ilaria Raciti
Paola Riva

11^{th} August 2017

Contents

1 Introduction

In Bayesian Statistics, a conventional way to sample from posterior distributions is to use Markov Chain Monte Carlo (MCMC) algorithms. However these methods can be computationally very expensive, especially if applied to *big data*. Indeed, not only the time per iteration and the time to reach convergence increase with the number of data, but it is usually impossible to process all data on a single machine. Moreover, even if data can be divided among machines, the communication costs during sampling are too high, regardless of the messages being passed (see Scott et al. [2016]).

To overcome these problems, an intuitive but promising approach to Bayesian Inference on Big Data consists in partitioning data into multiple subsets, called *shards*, store each of them on different machines and sample **independently** on each machine using only the corresponding shard.
The class of MCMC algorithms based on this logic is called *Embarrassingly Parallel* MCMC (EP-MCMC).
One of the main advantages of using EP-MCMC algorithms is the absence of communication among machines during the sampling phase: each machine draws MCMC samples using only its subset of data and, only afterwards, the result of sampling on each machine is combined to determine an estimate of the posterior distribution based on the entire dataset.

Existing EP-MCMC algorithms can be roughly divided into three types:

- algorithms based on asymptotic normality of posterior distribution, such as the well-known Consensus Monte Carlo algorithm by Scott et al. [2016];
- methods that recombine samples from subset posterior distribution by computing appropriate estimated summaries of the posterior; an example is proposed by Srivastava et al. [2015], which calculates the barycenter with respect to a Wasserstein distance between samples;
- algorithms relying on the so-called *product density equation* (PDE), that is a factorization of the posterior distribution.

These methods present some drawbacks.
The approach proposed in the Consensus Monte Carlo algorithm is effective when the posterior distributions are close to Gaussian, but could suffer from huge bias when skewness and multi-modes are present. In fact, bias effects that vanish in the full data, may be present in each shard and information asymmetry among machines may dramatically influence each subset's posterior.
The second and third category of EP-MCMC algorithms instead suffer respectively from lack of accuracy and inefficiency in re-sampling from the combined posterior.

We underline that *information asymmetry* is a common problem for these methods: if some of the *shards* do not contain information on the phenomenon that need to be studied, the estimates deriving from these shards could be far from the true distribution, leading to inaccurate posterior proxies.

Wang et al. [2015] proposed an EP-MCMC algorithm, termed PARALLEL AGGREGATION RANDOM TREES (PART), which aims to accomplish the solutions of the aforementioned problems.

Since PART algorithm belongs to the category of EP-MCMC algorithms based on PDE, we will now describe the assumptions of this class of methods.

Let us consider a dataset $X = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ of n Q-dimensional points and a d-dimensional parameter $\boldsymbol{\theta}$ of interest.
If

- there exists a partition $X^{(1)}, X^{(2)}, \ldots, X^{(M)}$ of X into M subsets such that $X^{(1)}, X^{(2)}, \ldots, X^{(M)}$ are conditionally independent on $\boldsymbol{\theta}$

$$p(X^{(1)}, X^{(2)}, \ldots, X^{(M)}|\boldsymbol{\theta}) = \prod_{m=1}^{M} f^{(m)}(X^{(m)}|\boldsymbol{\theta})$$

- the prior of $\boldsymbol{\theta}$ on the full data can be factorized as $\pi(\boldsymbol{\theta}) = \prod_{m=1}^{M} \pi^{(m)}(\boldsymbol{\theta})$

then the *product density equation* (PDE) holds:

$$p(\boldsymbol{\theta}|X) \propto \pi(\boldsymbol{\theta}) p(X|\boldsymbol{\theta}) \propto p(\boldsymbol{\theta}|X^{(1)}) p(\boldsymbol{\theta}|X^{(2)}) \ldots p(\boldsymbol{\theta}|X^{(M)})$$

where

- $p(\boldsymbol{\theta}|X)$, termed *full-posterior*, is the posterior distribution based on the entire dataset X
- $p(\boldsymbol{\theta}|X^{(m)})$ is the m^{th} *sub-posterior* representing the posterior distribution of $\boldsymbol{\theta}$ based on subset $X^{(m)}$, that can be written as

$$p(\boldsymbol{\theta}|X^{(m)}) \propto \pi^{(m)}(\boldsymbol{\theta}) f^{(m)}(X^{(m)}|\boldsymbol{\theta}).$$

If the above conditions hold, it is possible to sample **independently** from each sub-posterior $p(\boldsymbol{\theta}|X^{(m)})$, avoiding communication between different subsets. An estimate of the full-posterior distribution is obtained through an appropriate combination of samples drawn from each sub-posterior.

We underline again that the sampling steps are communication-free and only the combination phase requires communication between machines.

2 PART algorithm

The proposed algorithm, called Parallel Aggregation Random Trees (PART), is an EP-MCMC algorithm and, as such, its main goal is to give an estimate of the full-posterior as an aggregation of sub-posteriors.

The algorithm consists of two main steps, of which we will discuss the details in the following subsections:

1. *space partitioning*: partition the d-dimensional domain Θ of parameter $\boldsymbol{\theta}$,
2. *density aggregation*: aggregate the sub-posteriors' estimates to approximate the full-posterior.

2.1 Space Partitioning

The *space partitioning* phase of PART algorithm takes advantage of *random partition trees*, also referred to as *multi-scale histograms*.
Let us define a *rectangular block*

$$A_k := (l_{k,1}, r_{k,1}] \times (l_{k,2}, r_{k,2}] \times \cdots \times (l_{k,d}, r_{k,d}] \subseteq \mathbb{R}^d$$

and let $\mathcal{F}_K$ be the collection of all $\mathbb{R}^d$-partitions formed by K disjoint rectangular blocks.

The algorithm approximates each sub-posterior $p(\boldsymbol{\theta}|X^{(m)})$ using *K-block histograms*:

$$\hat{p}(\boldsymbol{\theta}|X^{(m)}) = \sum_{k=1}^{K} \frac{n_k^{(m)}}{N_m|A_k|} 1_{\boldsymbol{\theta}\in A_k}$$

where

- $n_k^{(m)}$ is the number of $\boldsymbol{\theta}$ samples drawn from machine m and falling into partition block A_k;
- N_m is the number of $\boldsymbol{\theta}$ samples drawn on machine m;
- K is the number of blocks that partition the domain Θ;
- $|A_k|$ indicates the area of the block A_k.

The first aim of PART algorithm is to determine a partition $\{A_1, ..., A_K\}$ of the domain Θ, where $A_1, ..., A_K$ and K are unknown and depend on the particular problem.

The partition is determined by recursively splitting each partition's block along a randomly selected dimension in $\{1, ..., d\}$, according to a predefined rule ϕ, according the the following procedure.
Let us consider a rectangular block $A := (l_1, r_1] \times (l_2, r_2] \times \cdots \times (l_d, r_d] \subseteq \mathbb{R}^d$ and suppose we want to split it along the randomly chosen but fixed dimension $p \in \{1, ..., d\}$. Denoting with $\{\boldsymbol{\theta}_j^{(m)}\}_{j\in A}$ those $\boldsymbol{\theta}$-samples drawn from the m^{th} sub-posterior and falling into partition's block A, the cutting point $\boldsymbol{\theta}_p^*$ along dimension p is determined through rule ϕ as follows:

$$\boldsymbol{\theta}_p^* = \phi(\{\boldsymbol{\theta}_{j,p}^{(1)}\}_{j\in A}, \{\boldsymbol{\theta}_{j,p}^{(2)}\}_{j\in A}, \ldots, \{\boldsymbol{\theta}_{j,p}^{(M)}\}_{j\in A})$$

where $\boldsymbol{\theta}_{j,p}^{(m)}$ is the p^{th} component of the j^{th} sample drawn on machine m that fall into partition's block A.
The resulting partition of block A along dimension p is given by:

$$A_p^- := [l_p, \theta_p^*)$$

$$A_p^+ := [\theta_p^*, r_p)$$

The bisection procedure is repeated on each of the resulting blocks $A^-, A^+ \subseteq \mathbb{R}^d$ until the block A_k satisfies one of the following stopping criteria:

- $\sum_{m=1}^{M} n_k^{(m)}/N_m < \delta_\rho$, which imposes a condition on the minimum number of samples falling into partition's block A_k;
- $|A_k| < \delta_a$, which guarantees a sufficiently wide area of the resulting partition's block A_k.

The partition rule ϕ defines how to perform the splitting along a fixed dimension p and can be chosen among one of the following two types:

- KD/median: the cutting point is the empirical median of samples falling into current block along dimension p;
- Maximum Likelihood: the cutting point corresponds to the maximizer of the empirical log-likelihood based on the projection along dimension p of samples falling into current partition block

$$\phi(\{\boldsymbol{\theta}_{j,p}^{(1)}\}_{j\in A}, \{\boldsymbol{\theta}_{j,p}^{(2)}\}_{j\in A}, \dots, \{\boldsymbol{\theta}_{j,p}^{(M)}\}_{j\in A}) =$$

$$= \underset{n_-+n_+=n, A^-\cup A^+=A}{\operatorname{argmax}} \sum_{m=1}^{M} \{n_-^{(m)}(\log n_-^{(m)} - \log|A^-|) + n_+^{(m)}(\log n_+^{(m)} - \log|A^+|)\}$$

where $n_- := \sum_{m=1}^{M} n_-^{(m)}$ and $n_+ := \sum_{m=1}^{M} n_+^{(m)}$ represent respectively the number of samples belonging to left A^- and right A^+ partition's block.

Since the computation cost of KD-rule is $\mathcal{O}(n)$ while ML-rule takes $\mathcal{O}(nlog(n))$, KD-Tree Partition is faster than ML one, especially when the number of posterior draws n in current block is large.

The bisecting procedure creates a *random partition tree* of the sampling domain: starting from a root node storing the entire domain Θ, at each node a dimension where to cut is *randomly* chosen and two children blocks are generated; the process is repeated until no further splitting is possible (i.e. one of stopping criteria has been met); in such a case the resulting node is a leaf of the tree; the set of leaves obtained represents a partition of the original domain Θ.

2.2 Density Aggregation

Once an estimate of each subset's posterior has been determined through *space partitioning*, PART algorithm combines all the sub-posteriors to determine an approximation of the full-posterior.

Each of the $m \in \{1, \dots, M\}$ sub-posterior estimate has the following form:

$$\hat{p}(\boldsymbol{\theta}|X^{(m)}) = \sum_{k=1}^{K} \frac{n_k^{(m)}}{N_m |A_k|} 1_{\boldsymbol{\theta} \in A_k}$$

where $\{A_k\}_{k=1}^K$ is a *rectangular partition* of Θ composed of K blocks.
Imposing the partition $\{A_k\}_{k=1}^K$ to be the **same** across all subsets, the estimated full-posterior is determined through the *product density equation* (PDE):

$$\begin{aligned}\hat{p}(\boldsymbol{\theta}|X) &= \frac{1}{Z} \prod_{m=1}^{M} \hat{p}(\boldsymbol{\theta}|X^{(m)}) = \\ &= \frac{1}{Z} \sum_{k=1}^{K} \left(\frac{1}{|A_k|^M} \prod_{m=1}^{M} \frac{n_k^{(m)}}{N_m} \right) 1_{\boldsymbol{\theta} \in A_k}\end{aligned}$$

where Z is the normalizing constant given by

$$Z = \sum_{k=1}^{K} \left(\frac{1}{|A_k|^{M-1}} \prod_{m=1}^{M} \frac{n_k^{(m)}}{N_m} \right).$$

Based on this result, new MCMC samples will be drawn from the estimated full-posterior density.

2.3 Improvements

We will now relax some of the previously introduced assumptions.

The first strong assumption to be relaxed concerns the shape of the full-posterior density, which, in general, can be written as a mixture of local generic densities as follows:

$$\hat{p}(\boldsymbol{\theta}|X) = \sum_{k=1}^{K} w_k g_k(\boldsymbol{\theta})$$

where w_k is the k^{th} mixture component's weight and $g_k(\boldsymbol{\theta})$ is a block-wise distribution.
The algorithm currently offers the following local kernels:

- uniform distribution: $g_k(\boldsymbol{\theta}) = Unif(\boldsymbol{\theta}; A_k)$ i.e. each sub-posterior is represented through *multi-scale histograms*;
- Gaussian distribution: $g_k(\boldsymbol{\theta}) = \mathcal{N}(\boldsymbol{\theta}; \mu_k, \Sigma_k)$, with μ_k and Σ_k estimates of the mean and covariance of samples in current partition's block A_k.

The use of local Gaussian kernels could help increasing the smoothness of the estimate.

Also notice that the assumption of a random partition of the space Θ that is **common** to all subsets is extremely restrictive, especially when the number of subsets M is large.
To overcome this issue, *pairwise aggregation* logic can be chosen: instead of considering all subsets together, *space partitioning* and *density aggregation* are

performed on each pair of subsets, until one final estimate of the full-posterior is obtained.

Finally, to achieve better approximation accuracy, the algorithm builds a *random forest* of n_{tree} independent random partition trees, each of which represents a possible partition of the domain Θ. The full-posterior estimate is then obtained as an average of the aggregated posteriors resulting from each random tree in the forest.

3 Implementation

We developed a `C++11` version of PART algorithm (available at `https://github.com/paolariva2/PART_BayesPACS`) taking as reference the original `MATLAB` code by *Wang et al.* (available at `https://github.com/richardkwo/random-tree-parallel-MCMC`).

3.1 PART

In the following sub-sections we will describe the implementation's details for each of the two main goals of PART algorithm:

1. *space partitioning*
   Determine a partition of K elements $\{A_1, ...A_K\}$ of the domain Θ, to approximate each sub-posterior $p(\boldsymbol{\theta}|X^{(m)})$ using a *K-block histograms*:

$$\hat{p}(\boldsymbol{\theta}|X^{(m)}) = \sum_{k=1}^{K} \frac{n_k^{(m)}}{N_m|A_k|} 1(\boldsymbol{\theta} \in A_k)$$

   where

   - $n_k^{(m)}$ is the number of $\boldsymbol{\theta}$ samples drawn from machine m and falling into partition block A_k;
   - N_m is the number of $\boldsymbol{\theta}$ samples drawn on machine m;
   - K is the number of blocks that partition the domain Θ;
   - $|A_k|$ indicates the area of the block A_k.

2. *density aggregation*
   Draw **new** samples from the estimated full-posterior, derived as aggregation of the estimated sub-posteriors $\{\hat{p}(\boldsymbol{\theta}|X^{(m)})|m \in \{1, \dots, M\}\}$ through *posterior density equation* (PDE):

$$\hat{p}(\boldsymbol{\theta}|X) \propto \prod_{m=1}^{M} \hat{p}(\boldsymbol{\theta}|X^{(m)})$$

3.1.1 Space Partitioning

As already explained, *space partitioning* consists in generating a random tree that partitions the domain Θ: starting from a root node storing the entire domain Θ, at each node a dimension where to cut is *randomly* chosen and two children blocks are generated; the process is repeated until no further splitting is possible (i.e. one of stopping criteria has been reached); in such a case the resulting node is a leaf of the tree; the set of leaves obtained represents a partition of the original domain Θ.

There are three main structures to be considered to grow a random partition tree:

- `RawMCMC`: reference class to store information on input subchains (see Listing 1).

Beside storing the number of subsets from which MCMC samples have been drawn (private attribute `m_M`) and the sampling area (private attribute `m_area`), an object of class `RawMCMC` concatenates all MCMC samples in a single matrix (private attribute `m_samples`) keeping track of the index of subset from which each of them have been drawn (private attribute `m_mark`).

Listing 1: include/RawMCMC.hpp

```
class RawMCMC{

    public:
        ...

    private:

        uword m_M;
        double m_sumLogN;
        mat m_samples;
        uvec m_mark;
        mat m_area;
        ...
};
```

- `MCestimate`: stores information on statistics, such as mean, covariance, sampling area and density estimate in logarithmic form associated to MCMC samples (see Listing 2).

  This class will be particularly important in re-sampling phase (see Section 3.1.2.1).

Listing 2: include/MCestimate.hpp

```
struct MCestimate{

    double log_prob;
    mat sampling_area;
    rowvec mean;
    mat Cov;

    MCestimate(const RawMCMC & rawMCMC,
               const uvec & idx, const mat & area);
};
```

- `Tree`: class to grow the random partition tree (see Listing 3).

  Its private attributes define how the tree should be grown:

  - `kd_cut`: if set to `true`, the cutting point at each partitioning phase will be determined through KD-method; otherwise, it will be computed as Maximum Likelihood (ML-method),

- `min_num_points_block`: minimum number of samples falling into each leaf node,
- `min_cut_length`: minimum length of sampling area along each dimension.

Listing 3: include/Tree.hpp

```
class Tree{

    public:
        ...

    private:

        bool kd_cut;
        uword min_num_points_block;
        double min_cut_length;

        struct Node{
            ...
        }

        void findCut(const RawMCMC & rawMCMC,
                     const Node & node,
                     double & cut,
                     uword & dim,
                     vec & samples_dim,
                     bool & valid_cut);

        bool checkCut(const double cut,
                      const vec & samples_dim,
                      const double l,
                      const double r);

        void build(const RawMCMC & rawMCMC,
                   Node & node,
                   const bool verbose,
                   std::vector<MCestimate> & leaves);
            ...
};
```

The tree is recursively grown calling the private method `build` on each node `Node` of the tree, until all leaves (represented through `std::vector<MCestimate> leaves`) have been determined.
Each node in the tree is represented through a private structure `Node`, storing relevant information to define the partition of sampling area (see Listing 4).

Listing 4: include/Tree.hpp

```
struct Node{

    Node * parent;
    Node * childLeft;
    Node * childRight;

    uvec indices; // indices of samples falling in node.
    mat area; // Sampling area.
    uvec dim_cut; // Candidate dimensions for cut.

    //! Constructor of root node
    Node();

    // Define attributes of child node and link it to parent
    void generateChild(const uword dim,
                       const double cut,
                       const double min_cut_length,
                       const bool is_left,
                       const uvec & idx,
                       Node & node);

};
```

The method `Tree::build` (see Listing 5) is responsible for assessing whether the considered `Node` node has to be a leaf ($valid\_cut == false$) or not. In the latter case a further partitioning will be applied to each of children nodes.

First of all the search of valid cutting point is performed among MCMC samples falling into current `node` through `Tree::findCut`.
If the selected cutting point is indeed valid ($valid\_cut == true$) the samples falling into current `node` need to be split (through `Node::generateChild`). Each child will then be submitted to `Tree::build` method as well.
Otherwise, if no valid cutting point have been found, the current node is indeed a leaf, representing one element of the space partitioning tree. In such a case, the statistics on samples falling into this leaf are stored in an object of class `MCestimate` and these information are added to the partition tree (represented through `std::vector<MCestimate>`).

Listing 5: src/Tree.cpp

```
void Tree::build(const RawMCMC & rawMCMC, Node & node,
                 const bool verbose,
                 std::vector<MCestimate> & leaves){

    ...

    // search for valid cutting point
```

```
        if(nMCsamples>min_num_points_block
           && node.dim_cut.n_rows>0){
            findCut(rawMCMC,node,cut,dim,samples_dim,valid_cut);
        }

        if(valid_cut){ // Valid partition has been found
            // define the partition into children nodes

            ...

            // create children of current node
            Node nodeLeft, nodeRight;
            node.generateChild(dim,cut,min_cut_length,true,
                               left_idx,nodeLeft);
            node.generateChild(dim,cut,min_cut_length,false,
                               right_idx,nodeRight);

            // proceed search on left child
            build(rawMCMC, nodeLeft, verbose, leaves);

            // proceed search on right child
            build(rawMCMC, nodeRight, verbose, leaves);

        }else{ // No valid partition has been found
            // this node is one element of the partition!!
            leaves.emplace_back(rawMCMC,node.indices,area);
        }

    };
```

To determine a valid cutting point, the private method `Tree::findCut` is called at the beginning of each call of `Tree::build`.

As shown in Listing 6, among all possible dimensions along which current area can be split (`node.dim_cut`), we randomly select only one of them (`dim`) and determine the cutting point through KD- or ML-method (according to `Tree.kd_cut`.
Once the candidate cutting point has been determined, the method `checkCut` defines if this cutting point indeed generates a valid partition, checking whether each candidate block is not too small and contains a sufficiently high number of samples.
If the candidate cut is valid ($valid\_cut == true$), the search stops in a leaf node.
Otherwise, the candidate dimension along which the invalid cutting point have been found is removed from the candidate dimensions along which the cut need to be searched and a new dimension is analyzed.
The call of `findCut` method terminates either with a valid cutting point along one dimension of sampling area or, after analyzing all available dimensions, with no valid cutting point.

Listing 6: src/Tree.cpp

```
void Tree::findCut(const RawMCMC & rawMCMC,
                   const Node & node, double & cut,
                   uword & dim, vec & samples_dim,
                   bool & valid_cut){

    valid_cut = false;

    // select candidate dimensions
    std::vector<uword> candidate_dims(
        conv_to< std::vector<uword> >::from(node.dim_cut) );

    while( (candidate_dims.size()>0) && (!valid_cut) ){
        // randomly select one dimension
        std::uniform_int_distribution<>
                        dis(0, candidate_dims.size()-1);
        const uword id_dim = dis(gen);
        dim = candidate_dims[id_dim];

        // select only samples of interest
        samples_dim = samples.col(dim);

        ...
        // select candidate cutting point according to KD or ML
        ...

        // check validity of cutting point
        valid_cut = checkCut(cut,samples_dim,
                             node.area(0,dim),
                             node.area(1,dim));

        candidate_dims.erase(
            candidate_dims.begin()+static_cast<int>(id_dim));

    }

    ...
};
```

We recall that, to relax the assumption of a common space partition among all subset's posterior, it is possible to grow a *random forest* instead of only one random tree. This means that it is possible to generate `ntree` trees to define the space partitioning (see Listing 7).

Listing 7: src/oneStageResampling.cpp

```
void oneStageResampling(const std::vector<const mat*> & subchains,
                        const Parameters & par,
```

```
                            const uword min_num_points_block,
                            mat & y){

        const RawMCMC rawMCMC(subchains);
        const uword nMCsamples = rawMCMC.get_samples().n_rows;
        const uvec dim_cut = find(
            rawMCMC.get_area().row(1)-rawMCMC.get_area().row(0)>
            par.min_cut_length
            );

        std::vector<std::vector<MCestimate> > forest(par.ntree);
        for(uword n=0; n<par.ntree; n++){

            // grow n^th random partition tree
            Tree tree(par.kd_cut,
                      min_num_points_block,
                      par.min_cut_length);

            tree.grow(rawMCMC,dim_cut, par.verbose, forest[n]);

        }

        posteriorResampling(forest,
                            par.n_samples,
                            par.gaussian_smooth,
                            par.verbose,
                            y);

    };
```

Since the cutting criteria KD and ML are deterministic and all trees in `forest` receive the same samples (`rawMCMC`) and the same dimensions along which the cutting point should be searched (`dim_cut`), if there is only one dimension along which the cutting point should be searched, the previously described procedure generates `ntree` identical trees.
To introduce randomness in growing trees, we perform an additional search step in `findCut` method: in such a case and only if a valid cutting point has been determined through previously described search, a new candidate cutting point is randomly selected among available samples; as before, the search continues until a valid cutting point is determined through `checkCut` (see Listing 8).

Listing 8: src/Tree.cpp

```
    if(nullptr==node.parent && 1==node.dim_cut.n_rows
       && valid_cut){

        dim=0;
        samples_dim = samples.col(dim);
```

```
            const double l = node.area(0,dim), r = node.area(1,dim);
            std::uniform_int_distribution<> dis(0,nMCsamples-1);
            valid_cut = false;
            while(!valid_cut){
                cut=samples_dim(dis(gen));
                valid_cut = checkCut(cut,samples_dim,l,r);
            }

        }
```

The result of the overall procedure is hence a `std::vector<MCestimate>` representing a rectangular partition $\{A_1, ...A_K\}$ of the domain Θ.

3.1.2 Density Aggregation

Once a partition of K elements $\{A_1, ...A_K\}$ of the domain Θ has been determined, it is necessary to define a way to aggregate the sub-posteriors to obtain the full-posterior estimate.
There are two logic to aggregate the estimated sub-posteriors:

- *one-stage aggregation*: given MCMC samples from all subsets, determine an estimate of the full-posterior aggregating **all** samples in one single step.
- *pairwise aggregation*: given MCMC samples from all subsets, determine an estimated posterior for each pair of sub-posteriors. Continue the aggregation procedure until a final estimate is obtained. The resulting estimated posterior is an approximation of the full-posterior.

As already said, the *pairwise aggregation* fashion is particularly useful to increase the accuracy of the full-posterior estimate.

Since the process of generating samples from the *aggregated posterior* does not depend on the choice of the aforementioned density aggregation logic, we first describe how to generate samples from an aggregated density and then discuss the details of the *pairwise* logic.

3.1.2.1 Posterior Resampling

To generate **new** samples from the *aggregated posterior*, the function `posteriorResampling` is called (see Listing 9).

First of all the indices of trees are randomly sampled for `ntree` times and the occurrences of each of them are stored in `tree_counted`.
Then, for each `tree` in `forest`, the leaves are sampled `tree_counted(ctree)`-times according to the probability associated to samples falling into the corresponding leaf (`prob`). In this way, `sampled_nodes` takes into account the relevance of both the selected `tree` and of its leaves.
Afterwards, for each unique value of previously sampled nodes, a leaf node is selected (`leaf`) and resampling is performed based on statistics stored in `leaf`. The number of samples drawn from `leaf` is proportional to the frequency of sampling `leaf` among leaves in current `tree` and the distribution from which the samples are drawn depends on the user's choice (i.e. *gaussian_smooth* == *true*

use local Gaussian; $gaussian\_smooth == false$ use uniforms).

We underline that, in order to actually draw samples from a local Gaussian distribution, it is not sufficient to set $gaussian\_smooth == true$. Indeed, if the covariance matrix of samples falling into current `leaf` is not invertible or not positive definite, a uniform kernel based on `leaf.sampling_area` is used instead.

Listing 9: src/posteriorResampling.cpp

```
void posteriorResampling(
        const std::vector<std::vector<MCestimate> > & forest,
        const uword n_samples,
        const bool gaussian_smooth,
        const bool verbose,
        mat & y){

    ...

    // number of MCMC samples drawn
    uword num_sampled = 0;

    for(unsigned int ctree=0;
        ctree<static_cast<unsigned int>(ntree);
        ctree++){

        // current tree
        const std::vector<MCestimate> tree(forest[ctree]);

        // weights of mixture components
        std::vector<double> prob(tree.size());
        unsigned int pos=0;
        for(auto const & i: tree){
            prob[pos] = std::exp(i.log_prob);
            pos++;
        }

        // how many times the current tree had been sampled (among ntree)
        const uword freq_tree = tree_counted(ctree);

        // Sample (with replacement) leaf nodes according to prob
        uvec sampled_nodes(freq_tree);
        std::random_device rd;
        std::mt19937 gen(rd());
        std::discrete_distribution<> discr(prob.begin(), prob.end());
        for(uword n=0; n<freq_tree; n++){
            sampled_nodes(n) = static_cast<uword>(discr(gen));
        }

        ...
```

```
            // for each unique value of sampled_nodes
            for(uword node_index = 0; node_index<n_unique; node_index++){

                // select leaf
                const unsigned int id =
                    static_cast<unsigned int>(unique_nodes(node_index));

                const MCestimate leaf = tree[id];

                // how many times current node had been sampled (in tree)
                const uword freq_node = nodes_count(node_index);

                // eigenvalue decomposition of leaf.Cov
                vec eigval;
                mat eigvec;
                if(gaussian_smooth && (leaf.Cov).is_finite()
                   && eig_sym(eigval, eigvec, leaf.Cov)
                   && eigval.is_finite() && eigvec.is_finite()
                   && all(eigval>0.0) ){

                    eigval = pow(eigval,0.5);

                    y.rows(num_sampled,num_sampled+freq_node-1) =
                        randn<mat>(freq_node,d)*diagmat(eigval)*(eigvec.t())
                        + repmat(leaf.mean,freq_node,1);

                }else{ // use uniform kernel

                    const rowvec L = leaf.sampling_area.row(0);
                    const rowvec R = leaf.sampling_area.row(1);

                    y.rows(num_sampled,num_sampled+freq_node-1) =
                        repmat(R-L,freq_node,1)%randu<mat>(freq_node,d)
                        + repmat(L,freq_node,1);

                }

                // number of samples drawn till now
                num_sampled += freq_node;

            } // end unique nodes for

        } // end tree for

};
```

3.1.2.2 Pairwise Aggregation

If the *pairwise aggregation* method is chosen for density aggregation, the func-

tion `pairwiseAggregation` is called (see Listing 10).

First of all the function defines an `unordered_map` to store MCMC samples from different subsets.
Then, until the `unordered_map` contains only one matrix, the following procedure is repeated:

- *matching*: define a way to create groups of subsets' indices.
  The idea is to associate pair of indices and to store each of them in `match`.
  To consider also the case of odd number of subsets, the last group of indices could contain three indices instead of a pair.
  The matching procedure is performed by `matchSubsets`, unless the number of samples' matrices is already three or less.
  The result of this procedure is `std::vector<std::vector<uword> > match`, whose g^{th} element stores the indices of subsets belonging to the g^{th} group.

- *group-posterior resampling*: for each group of matched indices, the corresponding samples are stored in `inner_subchains` to be elaborated by `oneStageResampling`.
  This function performs space partitioning on `inner_subchains` and then draws **new** samples from the corresponding aggregated posterior.
  The result of this *group-posterior resampling* procedure is a matrix storing MCMC samples drawn from the estimated posterior corresponding to the group of matched matrices.
  Since this matrix captures the properties of samples associated to g^{th} group, it will be used as input for the next stage of *pairwise aggregation*, while the previously used matrices associated to this group can be removed from `MCdraws`. Once the posterior resampling has been performed on all groups of indices in `match`, the object `MCdraws` has halved size and stores the posterior resampling result for each group.
  These matrices will be used to define the matching at next stage.
  After $\lfloor \log M / \log 2 \rfloor$ stages, where M is the number of input subsets, the object `MCdraws` contains only one matrix that corresponds to the desired full-posterior estimate.

Listing 10: src/pairwiseAggregation.cpp

```
void pairwiseAggregation(const Parameters & par,
                         const std::vector<mat> & subchains,
                         mat & aggr_post_samples){

    ...

    // initialize MCdraws
    std::unordered_map<uword,mat> MCdraws;
    for(uword i=0; i<M; i++){
        auto const it = MCdraws.emplace(i,subchains[i]);
    }

    while(MCdraws.size()>1){ // until only one matrix left
```

```
        // matching phase
        std::vector<std::vector<uword> > match;
        if(MCdraws.size()>3){

            matchSubsets(par.improve_matching, MCdraws,
                         par.verbose, match);

        }else{ // all subsets grouped together

            match.resize(1);
            for(auto const & i: MCdraws){
                match.front().push_back(i.first);
            }

        }

        // space partitioning and density aggregation on each group
        for(auto const & g: match){ // for each group in match

            const unsigned int n_matched = g.size();

            // store matrix associated to each matched subset
            std::vector<const mat*> inner_subchains(n_matched,nullptr);
            unsigned int pos=0;
            for(const uword i: g){
                auto const it = MCdraws.find(i);
                inner_subchains[pos] = &(it->second);
                pos++;
            }

            // group-posterior resampling result
            mat post_samples;
            oneStageResampling(inner_subchains,
                               par,
                               min_num_points_block,
                               post_samples);

            // update map content
            MCdraws.find(g[0])->second = std::move(post_samples);
            for(uword i=1; i<n_matched; i++){
                auto const it = MCdraws.erase(g[i]);
            }

        } // end resampling on each group

        ...

    } // only one matrix left
```

```
    // store estimate of full-posterior
    aggr_post_samples = MCdraws.begin()->second;

};
```

We said above that, if the number of matrices stored in `MCdraws` is more than three, the *matching* phase is performed calling the function `matchSubsets`. We will now describe more in detail how this function behaves.

There are two ways to perform the *matching* phase, according to the value of user-defined parameter `improve_matching` (see Listing 11).

The simplest case is $improve\_matching == false$, in which indices of subsets are paired without specific order. As previously said, if the number of input subsets is odd, the last index left out from matching will be stored in last group, which will contain three elements.

Observe that, by assumption, all *sub-chains* are independently generated and none of them is more relevant than the others. That is why, at the beginning of `matchSubsets`, we decided to randomly shuffle the indices of subsets: in such a way, at each execution of our code, the order of input subsets' indices collected from `MCdraws` changes.

If $improve\_matching == true$, pairs of subset indices will be formed according to the following metric:
subset's index $keyS$ is associated to $keyF$ if and only if it corresponds to the index of the median of squared distances between mean of samples. More specifically, if we call S_i and S_{keyF} the matrices of samples corresponding to i^{th} and $keyF^{th}$ indices respectively, $keyS$ is such that

$$sum\_dis\_sq(keyS) = median(sum\_dis\_sq)$$

where the i^{th} component of sum_dis_sq is defined as

$$sum\_dis\_sq(i) = \sum_{p=1}^{d}(\frac{1}{N_i}\sum_{n=1}^{N_i} S_i(n,:) - \frac{1}{N_{keyF}}\sum_{n=1}^{N_{keyF}} S_{keyF}(n,:))^2$$

with $S(n,:)$ indicating the n^{th} row of matrix S.

To implement this, we decided to use a `list`, named `subs_left_set`, into which we store the shuffled indices of subsets `id_sub`. Until the number of indices not yet matched is at most three, the following procedure is repeated:

1. select the first index of subset that has not yet been matched (`keyF`) and store the mean of the corresponding matrix in `meanF`;
2. for each element in `subs_left_set`, compute the sum of squared distances of samples' mean with respect to the selected subset and store the result in `sum_dis_sq` vector;

3. select the index of subset that corresponds to the median of `sum_dis_sq`, that is the $\lfloor subs\_left\_set.size()/2 \rfloor^{th}$ element of sorted indices of `sum_dis_sq` vector;

4. since the pair of indices has been determined, it is sufficient to store the indices in `match` and remove them from the set of indices that still need to be matched.

The procedure ends when there are at most three subsets left, that will be grouped together.

Listing 11: src/matchSubsets.cpp

```
void matchSubsets(const bool improve_matching,
                  const std::unordered_map<uword,mat> & MCdraws,
                  const bool verbose,
                  std::vector<std::vector<uword> > & match){

    // Number of subsets to be matched
    const uword subs_left = MCdraws.size();

    // Number of groups to be created
    const uword G = static_cast<uword>(std::floor(subs_left/2.0));

    match.resize(G, std::vector<uword>(2));

    // Randomness in matching
    uvec id_sub(subs_left);
    uword pos=0;
    for(auto const & it: MCdraws){
        id_sub(pos)=it.first;
        pos++;
    }
    id_sub = shuffle(id_sub);

    if(improve_matching){

        // indices of subsets to be matched
        std::list<uword> subs_left_set;

        for(const uword s: id_sub){
            subs_left_set.emplace_back(s);
        }

        uword n_groups = 0;

        // until no more than three subsets left out
        while(subs_left_set.size()>3){

            // first index of pair
```

```
            const uword keyF = *subs_left_set.begin();

            auto const itF = MCdraws.find(keyF);

            // mean of matrix associated to keyF
            const rowvec meanF = mean(itF->second,0);

            // store sum distances squared
            vec sum_dis_sq(subs_left_set.size());

            uword pos=0;
            for(const uword s: subs_left_set){

                auto const it = MCdraws.find(s);

                // mean distance w.r.t. keyF
                const rowvec dis(mean(it->second,0)-meanF);

                sum_dis_sq(pos) = accu(dis%dis);

                pos++;

            }

            const uvec id_sorted = sort_index(sum_dis_sq);
            const uword id_median = static_cast<uword>(
                            std::floor(subs_left_set.size()/2.0)));

            auto itS = subs_left_set.begin();

            // iterator to second index in current group
            std::advance(itS,
                         static_cast<int>(id_sorted(id_median)));

            // insert pair into match vector
            match[n_groups][0] = keyF;
            match[n_groups][1] = *itS;
            n_groups++;

            // remove form subs_left_set
            subs_left_set.erase(itS);
            subs_left_set.erase(subs_left_set.begin());
        }
        // subs_left_set.size() is at most three
        match[n_groups].resize(subs_left_set.size());
        std::copy(subs_left_set.begin(),subs_left_set.end(),
                  match[n_groups].begin());

    }else{ // match subsets without specific ordering
```

```
            // match pairs of indices
            for(uword g=0; g<G; g++){
                match[g][0]=id_sub(2*g);
                match[g][1]=id_sub(2*g+1);
            }

            // if odd number of subsets
            if(subs_left>2*G){
                match.back().push_back(id_sub(subs_left-1));
            }

    }
};
```

3.2 Interface

Since PART requires as input M independent subchains, we wrote an interface to generate them directly from data.

The abstract and virtual class `MCMC` and the associated struct `MCMCParams` allow the user to choose the desidered MCMC generator (see Listing 12).

The class `MCMC` is structured as follows:

- `m_subchains`: object into which the resulting subchains will be stored; this will be the input for PART algorithm.
- `generateSubchain`: method to generate the `indexChain`th subchain that will be stored in `m_subchains`, according to the Monte Carlo generator choosen by the user.
- `createInputSubchains`: method to split input data (saved in `pathInputData`) into M subsets and to generate corresponding subchains (calling `generateSubchain`).
- `get_subchains`: getter of protected attribute `m_subchains`.

Listing 12: interface/include/MCMC.hpp

```
//! Empty base struct for parameters of MCMC method
struct MCMCParams{
    virtual void f(){};
};

//! Base class for MCMC generation
class MCMC{

    protected:

      std::vector<mat> m_subchains;
```

```
    virtual void generateSubchain(const mat & subdata,
                                  const unsigned int indexChain) = 0;

  public:

    virtual void createInputSubchains(const std::string & pathInputData,
                                      const uword M,
                                      MCMCParams * parMC) = 0;

    inline const std::vector<mat> & get_subchains() const {
      return m_subchain;
    }

};
```

Based on the above structures, the user can define a derived class to generate MCMC samples using the desired Monte Carlo generator.
Since we decided to use *CmdStan* to generate each subchain, we defined the derived class `StanMC` and the corresponding structure `StanParams`.

The struct `StanParams`, derived from `MCMCParams`, is responsible for storing all parameters required to generate MCMC samples using *CmdStan*. Since *CmdStan* requires input data with extension *.data.R*, we also need to call *R* to transform each subdata in such format.

The attributes of `StanParams` hence take into account of parameters for *R* and *CmdStan* usage (see Listing 13):

- `m_samplingIter`: number of sampling iterations to be performed for each subchain;
- `m_burnin`: number of warmup iterations;
- `m_thin`: thinning parameter;
- `m_pathR`: path into which all *R* files will be found, namely *R* working directory path;
- `m_pathExeStan`: path of executable *Stan* model;
- `m_useInit`: `true` if initializations of sampler defined in external file.

These parameters are set from *.txt* file by the constructor of the structure (see Appendix A for default values)

Listing 13: interface/include/StanMC.hpp

```
struct StanParams: public MCMCParams{

    uword m_samplingIter;
    uword m_burnin;
```

```
    uword m_thin;
    std::string m_pathR;
    std::string m_pathExeStan;
    bool m_useInit;

    public:

      StanParams(){};

      StanParams(const std::string & fileMC);

      inline const uword get_samplingIter() const {
        return m_samplingIter;
      }

      inline const uword get_burnin() const {
        return m_burnin;
      }

      inline const uword get_thin() const {
        return m_thin;
      }

      inline const std::string & get_pathR() const {
        return m_pathR;
      }

      inline const std::string & get_pathExeStan() const {
        return m_pathExeStan;
      }

      inline const bool get_useInit() const {
        return m_useInit;
      }
};
```

An object of class `StanMC`, derived from `MCMC` class, is instead responsible for subchains' generation through *CmdStan*.

In particular, in addition to attributes and methods of base class `MCMC`, it also stores as private attribute a structure `StanParams` to easily access all MCMC parameters (see Listing 14)

Listing 14: interface/include/StanMC.hpp

```
class StanMC: public MCMC{

    StanParams m_parStan;
```

```
    void generateSubchain(const mat & subdata,
                          const unsigned int indexChain);

    public:

        StanMC(){};

        void createInputSubchains(const std::string & pathInputData,
                                  const uword M,
                                  MCMCParams * parMC);

        inline const StanParams & get_parStan() const {
            return m_parStan;
        }

};
```

We will now describe the details of the method to split data (`StanMC::createInputSubchains`) and the one to generate MCMC samples based on a given subset of data (`StanMC::generateSubchain`).

`Stan::createInputSubchains`

Since this method receives as input a pointer to `MCMCParams`, whereas we need to work on `StanParams`, we first perform a dynamic cast of the input pointer and then store the pointed value into private attribute `m_parStan` (see Listing 15). In such a way, we can easily access all parameters for MC generation.

Then the whole dataset, read from file `pathInputData`, is divided by rows into M subsets: each subset has `nRowsSubsets` rows, except the last one that contains all the remaining rows. This is the reason why the last subset is separately processed with respect to the first $M-1$.

Each subdata is then submitted to `Stan::generateSubchain` that fills `m_suchains` with the corresponding generated subchain.

Listing 15: interface/src/StanMC.cpp

```
void StanMC::createInputSubchains(const std::string & pathInputData,
                                  const uword M,
                                  MCMCParams * parMC){

    StanParams * ptrParStan = dynamic_cast<StanParams*>(parMC);
    m_parStan = *ptrParStan;

    // load input dataset
    mat data;
    if(!data.load(pathInputData)){
```

```
            throw std::logic_error ("Unable to read data file!!");
        }
        const uword n_data = data.n_rows;

        ...

        const uword nRowsSubset = static_cast<uword>(
                                  std::floor(n_data/static_cast<double>(M)) );

        m_subchains.reserve(M);

        // split data
        for(unsigned int m=0; m<M-1; m++){

            mat subdata(data.rows(0+m*nRowsSubset,(m+1)*nRowsSubset-1));

            generateSubchain(subdata,m);

        }

        // process all remaining rows
        mat subdata(data.rows((M-1)*nRowsSubset,data.n_rows-1));

        generateSubchain(subdata, M-1);

};
```

`Stan::generateSubchain`

This method, as shown in Listing 16, has two input parameters:

- `subdata`: matrix storing data onto which MCMC samples need to be drawn;
- `indexChain`: index of subchain to be generated.

As said above, *CmdStan* requires data to have particular extensions (*.data.R*) to the MC generation.
To transform data, we need to call the *R* function *stan_rdump*. We hence call an *R* script (`m_pathR`*/dataDump.R*) that transforms the content of *.csv* file storing `subdata` into *.data.R* format.

In a similar way, we proceed if the user decides to initialize the MC variables with custom values. In such a case, we generate a file with extension *.init.R* from `m_pathR`*/initDump.R*.

Once the subset of data and, eventually, the initialization have been written in compatible format with respect to *CmdStan*, we can call the MC generator with parameters defined in `m_parStan` attribute.
Since the output file generated through *CmdStan* contains headers and comments that *Armadillo* cannot process, we remove them and save only MCMC

samples in *.csv* format.

Now we can remove all unused files and easily read the result of *CmdStan* call, storing the generated chain in `indexChain`th position of `m_subchain`.

Listing 16: interface/src/StanMC.cpp

```
void StanMC::generateSubchain(const mat & subdata,
                              const unsigned int indexChain){

  const std::string pathR = m_parStan.get_pathR();
  const std::string nameSubdata(pathR + "/subdata.csv");
  if(!subdata.save(nameSubdata, csv_ascii)){
    throw std::logic_error ("!!! ERROR saving subdata !!! ");
  }

  const std::string cmdR_string ("Rscript "+ pathR + "/dataDump.R");
  const char * cmdR = cmdR_string.c_str();
  if(std::system(cmdR)!=0){
    throw std::logic_error ("!!! ERROR executing R session !!! ");
  }

  std::string cmdStan_string(m_parStan.get_pathExeStan() +
                             " sample num_samples=" +
                             std::to_string(m_parStan.get_samplingIter())
                             + " num_warmup=" +
                             std::to_string(m_parStan.get_burnin()) +
                             " thin=" +
                             std::to_string(m_parStan.get_thin()) +
                             " data file=" + pathR + "/subdata.data.R");

  if(m_parStan.get_useInit()){
    const std::string cmdRInit_string ("Rscript "+ pathR + "/initDump.R");
    const char * cmdRInit = cmdRInit_string.c_str();
    if(std::system(cmdRInit)!=0){
      throw std::logic_error ("!!! ERROR executing R session !!! ");
    }
    cmdStan_string = cmdStan_string + " init=" + pathR + "/inits.init.R";
  }

  const std::string nameDefaultChain (pathR + "/output.csv");
  cmdStan = cmdStan + " output " + nameDefaultChain;
  const char * cmdStan = cmdStan_string.c_str();
  if(std::system(cmdStan)!=0){
    throw std::logic_error ("!!! ERROR executing CmdStan session !!! ");
  }

  const std::string nameSubchain (pathR + "/subchain" +
                                  std::to_string(indexChain) + ".csv");
  const std::string cmdGrep_string("grep \"#\" -v "+ nameDefaultChain +
```

```
                                        " | grep -v lp__ > " + nameSubchain);
    const char * cmdGrep = cmdGrep_string.c_str();
    if(std::system(cmdGrep)!=0){
      throw std::logic_error ("!!! ERROR removing comments and header!!! ");
    }

    const std::string cmdRemove_string("rm " + nameSubdata + " " +
                                       pathR + "/subdata.data.R " +
                                       pathR + "/inits.init.R " +
                                       nameDefaultChain);
    const char * cmdRemove = cmdRemove_string.c_str();
    if(std::system(cmdRemove)!=0){
      throw std::logic_error ("!!! ERROR removing temporary files!!! ");
    }

    mat chain;
    if(!chain.load(nameSubchain)){
      throw std::logic_error ("!!! ERROR loading subchain !!! ");
    }
    m_subchains.push_back(chain);

};
```

4 Compile and Run

4.1 PART

The source code of our project is available for download at `https://github.com/paolariva2/PART_BayesPACS`.
To make use of our code, please install *Armadillo* library (see `http://arma.sourceforge.net/`). We used version `7.800.2` .

Before compiling our code, please modify according to your system the required environment variables in `setEnv.sh` file.

The following options are available in our `Makefile`:

compile in optimization mode (`-O2` flag)
```
> make
```

compile in debug mode (`-g -O0` flags)
```
> make debug
```

profile using `gprof` (`-pg` flag)
```
> make prof
```

generate documentation using `doxygen`
```
> make doxy
```

remove objects and executable
```
> make clean
```

restore original folder
```
> make distclean
```

Our program requires at least two input arguments to run, an could use an optional additional input:

1. Path of *.txt* file into which the paths of files storing subchains have been defined. Each subchain's file should have a compatible extension with respect to *Armadillo* (see `http://arma.sourceforge.net/docs.html#save_load_mat`).

2. Path of output file (`.csv`, no headers; see `csv_ascii` at `http://arma.sourceforge.net/docs.html#save_load_mat`), storing MCMC samples drawn from the aggregated posterior obtained through PART algorithm.

3. (OPTIONAL) Path of PART parameters file, defining parameters to be used in PART algorithm. This is a *.par* file for compatibility with *SigPack* (see `http://sigpack.sourceforge.net/` for additional details).
   In case this file is not given as input to the program, default parameters will be used, as defined in Appendix A.
   Recall that all parameters defined in this file must be coherent with the input subchains in order for the program to properly run.

To compile and run a simple example, proceed as follows:

```
> cd PART
```

```
> make

> ./main ../examples/data/d1/chains/M10/input_M10_N5.txt
../examples/data/d1/outPART_M10_N5_kdPairSmooth.csv
../examples/parameters/PART/M10/kdPairSmooth.par
```

For additional details on input files and how to run simple examples, please refer to `README.txt` and code documentation (`doc`/*DocumentationPART_Raciti_Riva.pdf*).

4.2 Interface

To make use of PART program interfaced with *CmdStan* and *R*, it is necessary to install *R* (`https://cran.r-project.org/`) and *RStan* (`https://github.com/stan-dev/rstan/wiki/Installing-RStan-on-Mac-or-Linux`). As MCMC generator, we used *CmdStan* (release `v2.16.0`), that is already available in `Interface/cmdstan-2.16.0` directory. For additional details on installation and usage of *CmdStan*, please refer to Appendix C.

To compile our program using interface, the user will find in `Interface` directory a *Makefile* that is similar to the one for PART program only.
As for compiling PART program, the user should *source* the file `setEnv.sh`, after appropriate setting of environmental variables, and then could use the same *make* options of PART (see Section 4.1).

PART program, interfaced with *CmdStan*, requires at least three inputs to run, and could use an optional additional input:

1. path of data file into which the entire dataset is stored.
   To read input data we use *Armadillo*, in particular `load` method of `Mat` class, that should automatically recognize file extensions (see `http://arma.sourceforge.net/docs.html#save_load_mat`).

2. path of PART output, storing MCMC samples from aggregated posterior in *.csv* extension (see `csv_ascii` at `http://arma.sourceforge.net/docs.html#save_load_mat`).

3. Path of *.txt* file storing information on MCMC parameters.
   In our case, this file must contain all information to run *CmdStan* and *R*, hence the following ordered inputs:

   (a) `samplingIter`: number of sampling iterations of MCMC algorithm;

   (b) `burnin`: number of burnin (warmup) iterations of MCMC algorithm;

   (c) `thin`: thinning parameter for MCMC algorithm;

   (d) `pathR`: path into which *R* scripts are located with respect to `Interface` directory. This path must be coherent with the working directory defined inside *R* scripts;

   (e) `pathExeStan`: path into which the executable of *Stan* model is located with respect to `Interface` directory (i.e. the one obtained compiling *.stan* model using *CmdStan*; see Appendix C);

(f) `useInit`: flag to establish if input file for initialization of parameters for MCMC sampling should be used; if set to `true`, the file `pathR`*/inits.init.R* will be used.

An example of the described file storing default values for *CmdStan* interface is available at Appendix C.

4. (OPTIONAL) Path of PART parameters file, defining parameters to be used in PART algorithm. This is a *.par* file for compatibility with *SigPack* (see `http://sigpack.sourceforge.net/` for additional details).
In case this file is not given as input to the program, the default parameters file will be used (see Appendix A for default parameter's file).
Recall that all parameters defined in this file must be coherent with the input subchains in order for the program to properly run.

To compile and run a simple example, proceed as follows:

```
> cd Interface

> make clean

> make

> ./main ../examples/data/d1/data.csv
../examples/data/d1/outPART_M10_N5_kdPairSmooth.csv
../examples/parameters/MCMC/paramsMCMC_N5.txt
../examples/parameters/PART/M10/kdPairSmooth.par
```

For additional details on input files and how to run simple examples, please refer to *README.txt* and code documentation (`doc`/*DocumentationInterfacePART_Raciti_Riva.pdf*).

5 Synthetic Tests

Using *R* and *CmdStan* (see Appendix C), we generated a synthetic dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ to test our implementation of PART algorithm.
We simulated $n = 2000$ observations according to the following model:

$$Y_i \mid (\mathbf{x}_i, \beta_0, \boldsymbol{\beta}) \sim Be(q_i) \qquad i = 1, \dots, n$$

$$logit(q_i) = \beta_0 + \boldsymbol{\beta}\, \mathbf{x}_i^T \qquad i = 1, \dots, n$$

$$\mathbf{X}_i \sim \mathcal{N}_d(\mathbf{0}, \boldsymbol{\Sigma}) \qquad i = 1, \dots, n$$

$$\Sigma_{k,l} = 0.9^{|k-l|} \qquad k, l = 1, \dots, d$$

$$\beta_0 = -3\,, \qquad \beta_1, \dots, \beta_d \overset{i.i.d.}{\sim} \mathcal{N}(0, 25)$$

Then we randomly split the above dataset $\{(\mathbf{x}_i, y_i)\}_{i=1}^n$ into M equally sized subsets and we used each of them to independently generate N MCMC draws for each subset.

We performed a variety of tests considering different combinations of the following parameters:

- dimension d of the parameter vector $\boldsymbol{\beta}$ ($d \in \{1, 9, 29\}$);
- number N of MCMC samples stored ($N \in \{5000, 10000, 25000\}$). These samples were all generated setting $burnin = 1000$ and $thin = 4$;
- number of subsets M into which original dataset need to be partitioned ($M \in \{10, 20, 40\}$);
- parameters of PART algorithm:
  - `pairwise_aggregation`: `true` if *pairwise* aggregation is used to perform density aggregation, `false` if *one-stage* choosen;
  - `improve_matching`: `true` if matching performed according to MCMC samples, `false` if no specific association order required;
  - `halve`: `true` if minimum fraction of samples in each leaf need to be halved at each stage of *pairwise* aggregation, `false` if same value used at each stage;
  - `ntree`: number of random partition trees to be grown;
  - `kd_cut`: `true` if KD cutting type need to be used, `false` if ML cut type choosen;
  - `min_frac_points_block`: minimum fraction of samples in each leaf node,
  - `min_cut_length`: minimum length of partition's block along one dimension,

- `resample_N`: number of samples to be drawn from aggregated posterior,
- `gaussian_smooth`: `true` if aggregated posterior is a mixture of local Gaussian, `false` if mixture of local uniforms.

To test the accuracy of PART algorithm we considered:

$\boldsymbol{\beta}^{true}$ vector of parameters that we used to generate the synthetic dataset,

$\{\hat{\boldsymbol{\beta}}^{(g)}\}_{g=1}^{N}$ samples deriving from the full-posterior, that is the one based on the whole data,

$\{\boldsymbol{\beta}^{*(g)}\}_{g=1}^{N}$ samples deriving from aggregated posterior estimate obtained through PART.

We choose the following metrics to test the accuracy of PART estimates:

- Root-Mean-Square Error (RMSE) of PART estimates w.r.t. samples from the full-posterior:

$$rmse(\{\boldsymbol{\beta}^{*(g)}\}_{g=1}^{N}) = ||\frac{1}{dN}\sum_{g=1}^{N}(\boldsymbol{\beta}^{*(g)} - \hat{\boldsymbol{\beta}}^{(g)})||$$

  This quantity measures the accuracy of PART in approximating the full-posterior density.

- Posterior Concentration Ratio (r):

$$r^2 = \frac{\sum_{g=1}^{N}||\boldsymbol{\beta}^{*(g)} - \boldsymbol{\beta}^{true}||^2}{\sum_{g=1}^{N}||\hat{\boldsymbol{\beta}}^{(g)} - \boldsymbol{\beta}^{true}||^2}$$

  The closer the posterior concentration ratio is to the unit value, the closer the estimates of PART are with respect to the full-posterior's one. If this value is close to one from below, PART's estimate is even better than the full-posterior in approximating the true distribution of parameters vector $\boldsymbol{\beta}$.

In this section we first describe the performances of our implementation of PART, considering default values of its parameters (see Appendix A).
Later on, we will discuss in detail the effects of varying these parameters in both *space partitioning* and *density aggregation* steps. These tests are particularly useful to evaluate the consequences of choosing inappropriate values of PART parameters.

5.1 Accuracy and Times

The first analysis focuses on the accuracy and execution time of PART algorithm with default PART parameters, as described in Appendix A.

We considered different scenarios:

- dimension d of the parameter vector $\boldsymbol{\beta}$ ($d \in \{1, 9, 29\}$);

- number N of MCMC samples stored ($N \in \{5000, 10000, 25000\}$). These samples were all generated setting $burnin = 1000$ and $thin = 4$;
- number of subsets M into which original dataset need to be partitioned ($M \in \{10, 20, 40\}$);

A detailed list of results for each of these combinations is available at Appendix B (Tables 5 - 16, Figures 10 - 12).

First of all we observed that, for any fixed dimension d of $\boldsymbol{\beta}$, the range of *RMSE* and *posterior concentration ratio* values is quite small, indicating that all PART's configurations behave similarly to each other.
Since the *RMSE* values are small and the *posterior concentration ratio* values are close to the unit value, we conclude that PART is able to give accurate estimates not only of the full-posterior density but also of the true distribution of $\boldsymbol{\beta}$ parameters.

Moreover we noticed that the PART's configurations usually achieving better results correspond to *pairwise aggregation* and *gaussian smoothing*, coherently with the Gaussian priors of the considered model.
Also observe that the choice of Maximum Likelihood (ML) cutting rule for partitioning could lead to better proxies.

We also underline that, in general, increasing the number of subsets M does not guarantee better results. This is a consequence of the *information asymmetry* problem from which all methods based on PDE suffer (see Section 1).

Looking at the execution times, we noticed that, for fixed dimension d and number of subsets M, the most expensive case is always associated to the *one-stage aggregation* logic. Indeed this aggregation type combines all the matrices of MCMC samples together before partitioning, resulting in huge memory requirements and high computational costs.

Observe also that Maximum Likelihood (ML) always takes more time than KD-median cutting rule, coherently with the small computational costs of KD at each node of partition tree, as described in Section 2.1.
We can also see that, for a fixed dimension d of the problem, the execution time of ML configurations has non-linear growth in the number of subsets M, while this increase is not so marked in KD cases.

From these results, we conclude that our implementation of PART is able to achieve good results in short times.

The above tests also suggest to:

- properly choose the number of subsets M into which data should be divided, to reduce the effect of *information asymmetry*;
- prefer *pairwise* to *one-stage* aggregation, since the latter always requires higher memory consumption and computational costs without significant improvements in terms of accuracy of results;

- consider using Gaussian kernels, according to the form of prior distributions.

We will now focus our attention on the scenario with $d = 9$ components of $\boldsymbol{\beta}$ vector and $N = 5000$ MCMC samples stored on each subset, discussing the details of tests performed on each of the two main steps of PART algorithm: *space partitioning* and *density aggregation*.

5.2 Space Partitioning

First of all observe that PART usually achieves better results with $M = 10$ subsets, coherently with the dimensions of our dataset and with the *information asymmetry* problem, discussed in Section 1.

In the following tests, we decided to monitor not only the overall execution time of our program, but also the partial time spent on *space partition*, hence noticing that the most of execution time is indeed spent growing the random partition forest (see also profiling, Appendix B).

We will now show the influence of the following parameters on *space partitioning*:

- `min_frac_points_block`: minimum fraction of samples falling into each partition's block w.r.t. the original number of samples,
- `min_cut_length`: minimum partition's block area along each direction,
- `ntree`: number of random partition trees to be grown.

We simply recall that the first two parameters are related to the stopping criteria for growing a tree (Section 2.1).

`min_frac_points_block` (Appendix B: Tables 17 - 19)

In the considered scenario, if we choose $min\_frac\_points\_block = 0.001$, the search of a valid cutting point could continue until there are just 5 samples in each partition's block. Because of this, we obtain a large number of leaves - around 1800 - and inaccurate estimates of the posterior density - quite high *RMSE* and *posterior concentration ratio* values.

On the other hand, with $min\_frac\_points\_block = 0.1$, the minimum number of samples in each partition's block is restricted to be no less than 500. This generates in very short time a pruned random partition tree with very few leaves - around 15 - corresponding to wide partition's block and leading to bad proxies.

Choosing instead an intermediate value for the above parameter ($min\_frac\_points\_block = 0.01$), we obtain more balanced results: since the minimum number of points in each partition's block is 50, the estimated posterior is quite close to the full-posterior (small $RMSE$ values) and closer to the true density (r values close to 1 from below).

`min_cut_length` (Appendix B: Tables 20 - 22)

We noticed that, usually, the more this parameter increases, the worse the accuracy of estimates becomes.
Indeed, in general, imposing large values of *min_cut_length*, is equivalent to decide *a priori* how wide the partition's block will be.

By default, we choose to keep this parameter low, to avoid imposing too wide areas. In such a way, the more relevant stopping criterion in growing the random partition tree will be the one based on the number of samples in each block, that is a data-driven condition.

We hence suggest the user to properly set the value of this parameter, according to the specific model.

`ntree` (Appendix B: Tables 23 - 25)

Finally we analyzed the effect of growing a forest of different dimensions.

As expected, the accuracy of results decreases with the number of random partition trees in the forest (see Section 2.2). Since with 16 trees we already achieve good estimates in low times and the accuracy improvement using 32 trees is not significant, we choose $ntree = 16$ as reference value.

We underline here that the growth of the random forest is one of the *embarassingly parallelizable* phases of PART algorithm, and so, introducing a simple parallelization, it would be possible to significantly reduce the overall execution time.

5.3 Density Aggregation

We focus now on the analysis of *density aggregation* performed with *pairwise* logic, discussing the role of the following parameters:

- `improve_matching`: whether to match subsets according to properties of stored samples or to randomly associate them without a specific criterion,
- `halve`: whether to halve the fraction of samples that could fall into each partition's block at each stage of pairwise aggregation,
- `resample_N`: number of samples drawn from the aggregated posterior.

`improve_matching` (Appendix B: Table 26)

As described in Section 3.1.2.2, if this parameter is set to `true`, at each stage of *pairwise aggregation* the matching among samples is based on the following metric: subset's index $keyS$ is associated to index $keyF$ if and only if the latter corresponds to the median of squared distances between mean of samples.

More specifically, if we call S_i and S_{keyF} the matrices of samples corresponding to i^{th} and $keyF^{th}$ indices respectively, $keyS$ is such that

$$sum\_dis\_sq(keyS) = median(sum\_dis\_sq)$$

where the i^{th} component of sum_dis_sq is defined as

$$sum\_dis\_sq(i) = \sum_{p=1}^{d}(\frac{1}{N_i}\sum_{n=1}^{N_i} S_i(n,:) - \frac{1}{N_{keyF}}\sum_{n=1}^{N_{keyF}} S_{keyF}(n,:))^2$$

with $S(n,:)$ indicating the n^{th} row of matrix S.

We observed that, imposing $improve\_matching == true$, the accuracy of PART estimates actually improves: this configuration achieves lower $RMSE$ and values of *posterior concentration ratio* that are closer to the unit value from below than the one with $improve\_matching == false$. This improvement is however paid in terms of execution time: even if low, the execution time is almost doubled with respect to $improve\_matching == false$ case.

`halve` (Appendix B: Tables 27, 28)

When $halve == true$, *pairwise aggregation* has an additional feature, according to which the minimum fraction of samples falling into each partition's block is halved at each stage. To keep as minimum the input value of $min\_frac\_points\_block$, the initial value of this parameter is set to $min\_frac\_points\_block * 2^{\lfloor \log M / \log 2 \rfloor}$, so that the value at last stage is indeed $min\_frac\_points\_block$.

We studied the performances of PART algorithm with *pairwise aggregation* comparing the case of minimum fraction of samples halved at each stage and the one in which $min\_frac\_points\_block$ always assumes the same value.

Imposing as input a small value of *min_frac_points_block* (0.001), we achieve better results choosing $halve == true$.
This is coherent with results on the influence of *min_frac_points_block*: choosing the halving condition, at first stage we obtain a minimum fraction of samples that is $min\_frac\_points\_block = 0.001 * 2^{\lfloor \log M / \log 2 \rfloor}$. Halving it at each stage, the last aggregation step will have $min\_frac\_points\_block = 0.001$.
In such a way, the constraint on minimum number of samples at first stage is not too restrictive - 40 in the considered case - but it is further halved at each stage.

Instead observe that the halving condition is not helpful if the input minimum fraction of points is already sufficiently high: choosing the default value of $min\_frac\_points\_block$ (0.01), there is no advantage in halving this parameter at each stage. This is why, by default, we choose the combination $halve = false$, $min\_frac\_points\_block = 0.01$.

`resample_N` (Appendix B: Table 29)

This parameter represents the number of samples drawn from each aggregated posterior.

We studied the behaviour of the accuracy of estimates varying *resample_N* between 1500 and 15000.
Even if the accuracy values are here quite close to each other, we cannot assess the existence of a specific relationship between accuracy measures (either *RMSE* or *posterior concentration ratio*) and the number of posterior samples.
Observe, however, that KD already achieves good results with almost 5000 posterior samples, while ML reaches the best level of accuracy around 10000 samples.

According to these results, we decided to set 10000 as default value for the number of posterior samples.

6 Application

We decide to apply the PART algorithm to a recommender system problem.
Recommender systems are information filtering techniques that seek to predict the preferences of users: retailers use this kind of technologies to recommend new items to their customers, based on what they have bought in the past, and, as a consequence, to increase their satisfaction.
There exist mainly two types of recommender systems:

- *content-based filters* that recommend to each user items that are similar to those that he has liked in the past;
- *collaborative filters* that identify users with similar tastes and recommend them items that they liked.

Content-based filtering approach can be applied when information about the items is available, whereas collaborative filtering usually needs users' features.

6.1 Reference Model

Condliff et al. [1999] proposed a Bayesian model for recommender systems that incorporates all the available information: user ratings, user features and item features.
Let us go into the details of this model.
Suppose to have N users, each having a feature vector of length Q, and M items, each described by a feature vector of length K.
Both user covariates vectors $\{\mathbf{x}_i\}_{i=1:N}$ and item features vectors $\{\mathbf{f}_j\}_{j=1:M}$ are binary vectors whose elements are 1 if the feature is present and 0 if not:

$$x_{is} = \begin{cases} 1 & \text{if user } i \text{ has covariate } s \\ 0 & \text{otherwise} \end{cases}$$

$$f_{jt} = \begin{cases} 1 & \text{if item } j \text{ has feature } t \\ 0 & \text{otherwise} \end{cases}$$

Each user rated each item. A $N \times M$ binary matrix L collects those ratings: each element L_{ij} is the rating that user i gave to the item j, that is 1 if he likes the item and 0 if he does not like it.
Appealing to the idea that when we are looking for suggestions we seek the advice of friends with similar tastes, the model allows users to borrow strength from each other. Moreover, the model goes deeper, since it also takes advantage of the knowledge of the features of all the items.

The goal is to compute for each item j described by the feature vector $\mathbf{f}_j$ the probability that a user i will like it, and then to recommend to the user those items with the highest probabilities.Thanks to the Bayes Theorem, we can compute it as:

$$P(L_{ij} = 1|\mathbf{f}_j) = P(L_{ij} = 1)\frac{P(\mathbf{f}_j|L_{ij} = 1)}{P(\mathbf{f}_j)}.$$

The odds ratio of this probability is

$$odds_{ij} = \frac{P(L_{ij}=1|\mathbf{f}_j)}{P(L_{ij}=0|\mathbf{f}_j)} = \frac{P(L_{ij}=1)}{P(L_{ij}=0)}\frac{P(\mathbf{f}_j|L_{ij}=1)}{P(\mathbf{f}_j|L_{ij}=0)}.$$

Take the logarithm:

$$\log(odds_{ij}) = \log\Big(\frac{P(L_{ij}=1)}{P(L_{ij}=0)}\Big) + \log\Big(\frac{P(\mathbf{f}_j|L_{ij}=1)}{P(\mathbf{f}_j|L_{ij}=0)}\Big).$$

Then, assuming for each user that item features are conditionally independent given the user's rating:

$$\log(odds_{ij}) = \log\Big(\frac{P(L_{ij}=1)}{P(L_{ij}=0)}\Big) + \sum_{k=1}^{K}\log\left(\frac{P(f_{jk}=1|L_{ij}=1)}{P(f_{jk}=1|L_{ij}=0)}\right).$$

Now, assume that the logit of the the probability that item j, which user i likes, has feature k is

$$logit\big(P(f_{jk}=1|L_{ij}=1)\big) = \mu_{ik} + \psi_{ik} + \delta_{ik} \tag{1}$$

and the logit of the probability that item j, which user i does not like, has feature k

$$logit\big(P(f_{jk}=1|L_{ij}=0)\big) = \mu_{ik} + \psi_{ik} \tag{2}$$

where

μ_{ik} is the feature-level random effect,

δ_{ik} is the incremental effect of liking the item,

ψ_{ik} is the effect of the user characteristics.

The priors assigned are:

$$\mu_{ik} \sim \mathcal{N}(0, 100000)$$
$$\beta_{sk} \sim \mathcal{N}(0, 100000)$$
$$\delta_{ik} \sim \mathcal{N}(d_{ik}, \frac{1}{\tau_{ik}})$$
$$d_{ik} \sim \mathcal{N}(0, 100000)$$
$$\tau_{ik} \sim \Gamma(0.001, 0.001).$$

Moreover, the distribution of $\{P(L_{ij})\}_{i=1:N,j=1:M}$ is assumed to be a beta.

6.2 Book Crossing Dataset

6.2.1 Data

The dataset, which we want to apply a recommender system to, is the *Book Crossing* dataset.
This dataset was collected by Ziegler et al. [2005], in September 2004, from the Book-Crossing community. The Book-crossing community is an online book

club: each member leaves a book in a public place to be picked up and read by other members, who then do likewise. When a book enters in the system, the original owner describes it by assigning some labels, and then the book is tracked during its travel around the world: every time a member finds and reads a book, he gives a rating to it.
The dataset contains 278858 users providing 1149780 ratings about 271379 books.

In particular, it comprises three tables:

- users' details: each user is anonymized and mapped to an ID, but he provides information about its age and the city, region and country where he lives;
- books' details: each book comes with the information about the title, the author, the year of publication and the publisher.
- ratings' details: each rating is associated to a unique user and a unique book.

The ratings ranges between 0 and 10, where 0 means completely disliked and 10 very appreciated. Notice that not all users rated all books: if a user has not read a book, then his rating for that book is missing.

Many demographic and age information is missing in the dataset: we decide to deal only with complete data and to remove those users that did not provide their details. Moreover, we notice the presence of meaningless data referred to the age of users: we delete those users who said to be 0 years old or more than 100 years old.

Furthermore, we realize that the year-of-publication label associated to books is not related to the first edition of the book, but to each particular copy. Thus, this variable seems not very informative and characterizing.
A much more useful information to portrait a book would be its genre. In order to retrieve the genre of each book, we accessed an API: an API is an interface to access resources through a web browser. In particular, we use Google Books repository to integrate our data: thanks to Google Books API, it is possible to perform searches based on the identification code of a book and retrieve many information about it.
The identification code used in the *Book Crossing* dataset is the ISBN-10, which is a 10 digits identifier, whereas books in Google Books repository are associated to the ISBN-13, which is a 13 digits identifier; thus we convert all the ISBN-10 into ISBN-13.
For each book in the *Book Crossing* dataset, we make a query and retrieve its genre and the number of pages. In Appendix D the code to get these information is reported. Then, we merge the original book's details with the result from Google Books. As a result, only a subset of books actually get a match and are associated to a genre.

We transform and store the users' details and books' details the following form: X is a $N \times Q$ matrix whose rows are the user covariates binary vectors

and F is a $M \times K$ matrix whose rows are the book features binary vectors.

$$X = \begin{bmatrix} \mathbf{x}_1 \\ \mathbf{x}_2 \\ \dots \\ \mathbf{x}_N \end{bmatrix}, \quad \mathbf{x}_i = \begin{bmatrix} x_{i1} & \dots & x_{iQ} \end{bmatrix}, \quad x_{iq} = \begin{cases} 1 & \text{if user } i \text{ has covariate } q \\ 0 & \text{otherwise} \end{cases}$$

$$F = \begin{bmatrix} \mathbf{f}_1 \\ \mathbf{f}_2 \\ \dots \\ \mathbf{f}_M \end{bmatrix}, \quad \mathbf{f}_j = \begin{bmatrix} f_{j1} & \dots & f_{jK} \end{bmatrix} \quad f_{jk} = \begin{cases} 1 & \text{if book } j \text{ has feature } k \\ 0 & \text{otherwise} \end{cases}.$$

The ratings are transformed as a $N \times M$ matrix R whose elements r_{ij} represents the rating that user i gave to book j, it this rating exists, otherwise it is a null value, indicating that user i did not read book j.
In order to have the binary *Like* variable, we also dichotomize the ratings and create a $n \times M$ matrix L whose elements are

$$L_{ij} = \begin{cases} 1 & \text{if } r_{ij} \geq 6 \\ 0 & \text{if } r_{ij} < 6. \end{cases}$$

The rating matrix R is highly sparse. In order to avoid the risk of having a too non-informative scenario, we decide to retain only those users who rated at least 3 books and those books who have been rated by at least 3 people.

Since the world-wide publishers considered in the *Book Crossing* dataset are 16785 and we do not know most of them, we decide to keep the *publisher* label only for books rated by Italian users. In this way, we can better interpret the results.
In particular, we consider two different set of data, which we will refer to as *World-wide Book Crossing* dataset and *Italian Book Crossing* dataset.

***World-wide Book Crossing* dataset**
The dataset consists of:

- 1860 users,
- 574 books,
- 14629 ratings.

The users' covariates that we use are the age and the continent (see Figure 1). The book's features, reported in Figure 2, are the information we integrated thanks to Google Books APIs: the genre and the number of pages.
We code the users's covariates and books's features as the indicators reported in Appendix E.

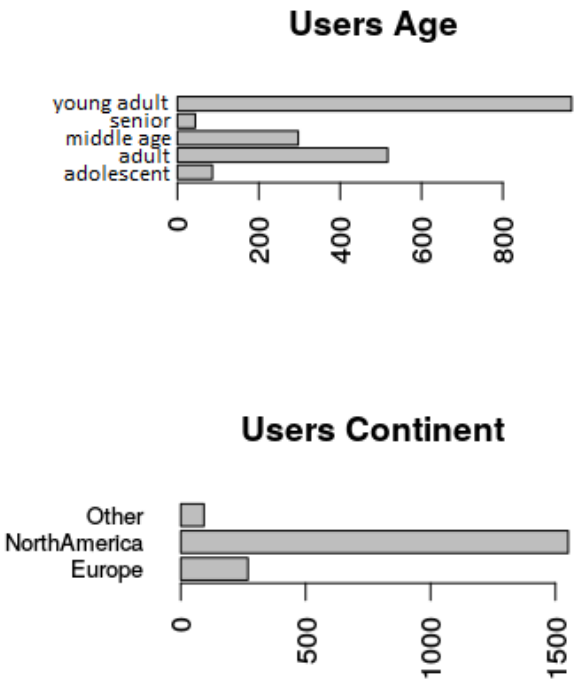


Figure 1: Bar-plot of world-wide users covariates.

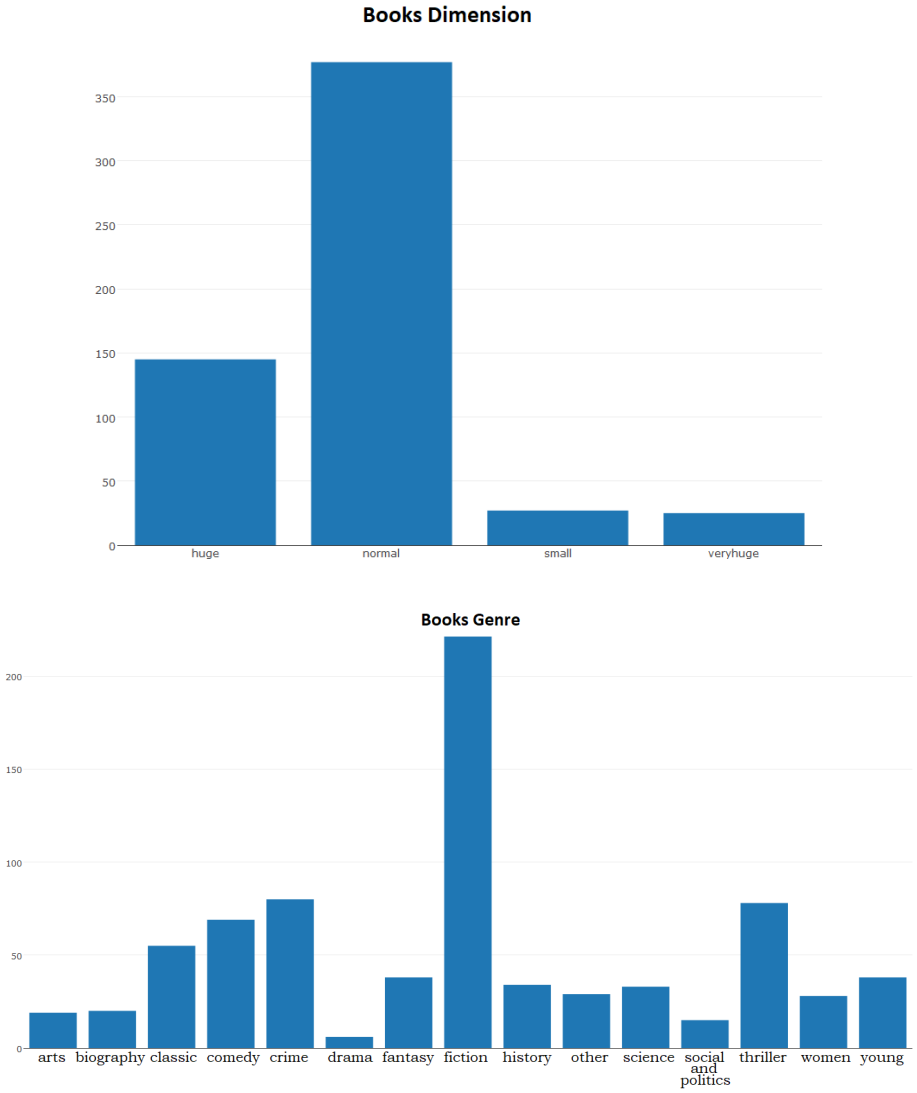


Figure 2: Bar-plot describing the features of books rated by world-wide users.

***Italian Book Crossing* dataset** We filter users according to their nationality, and we keep only Italian users and books rated by them.
The resulting dataset consists of:

- 238 users,
- 143 books,
- 628 ratings.

The users' covariates that we use are the age, the position in Italy of the residence, the information about the city of residence, if it an administrative center or not. In Figure 4 we report the bar-plots representing the Italian users' covariates.
The book's features we consider are the genre and the publisher.
The indicators coding the users's covariates and books' features are reported in Appendix E.

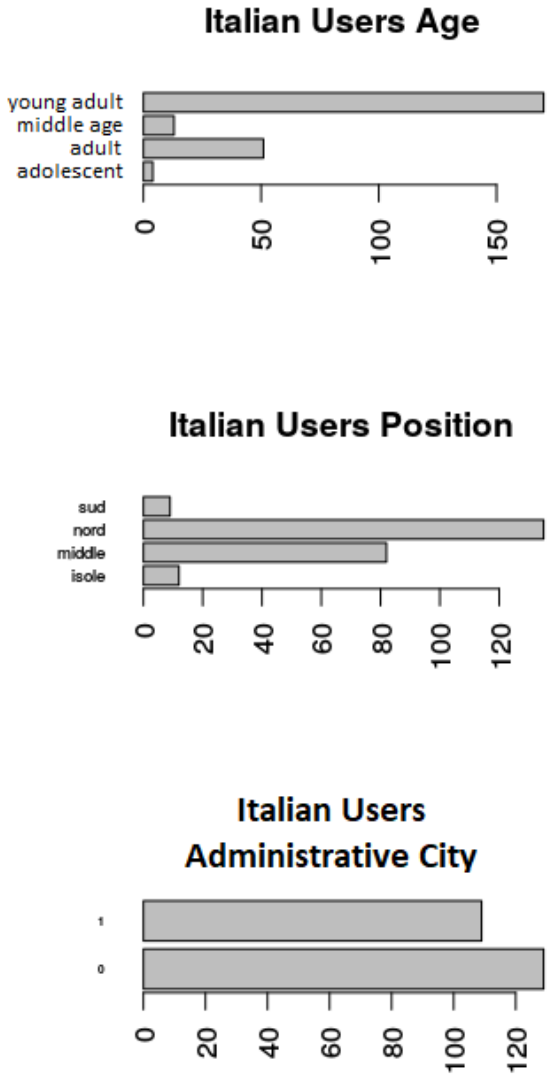


Figure 3: Bar-plot of Italian users covariates.

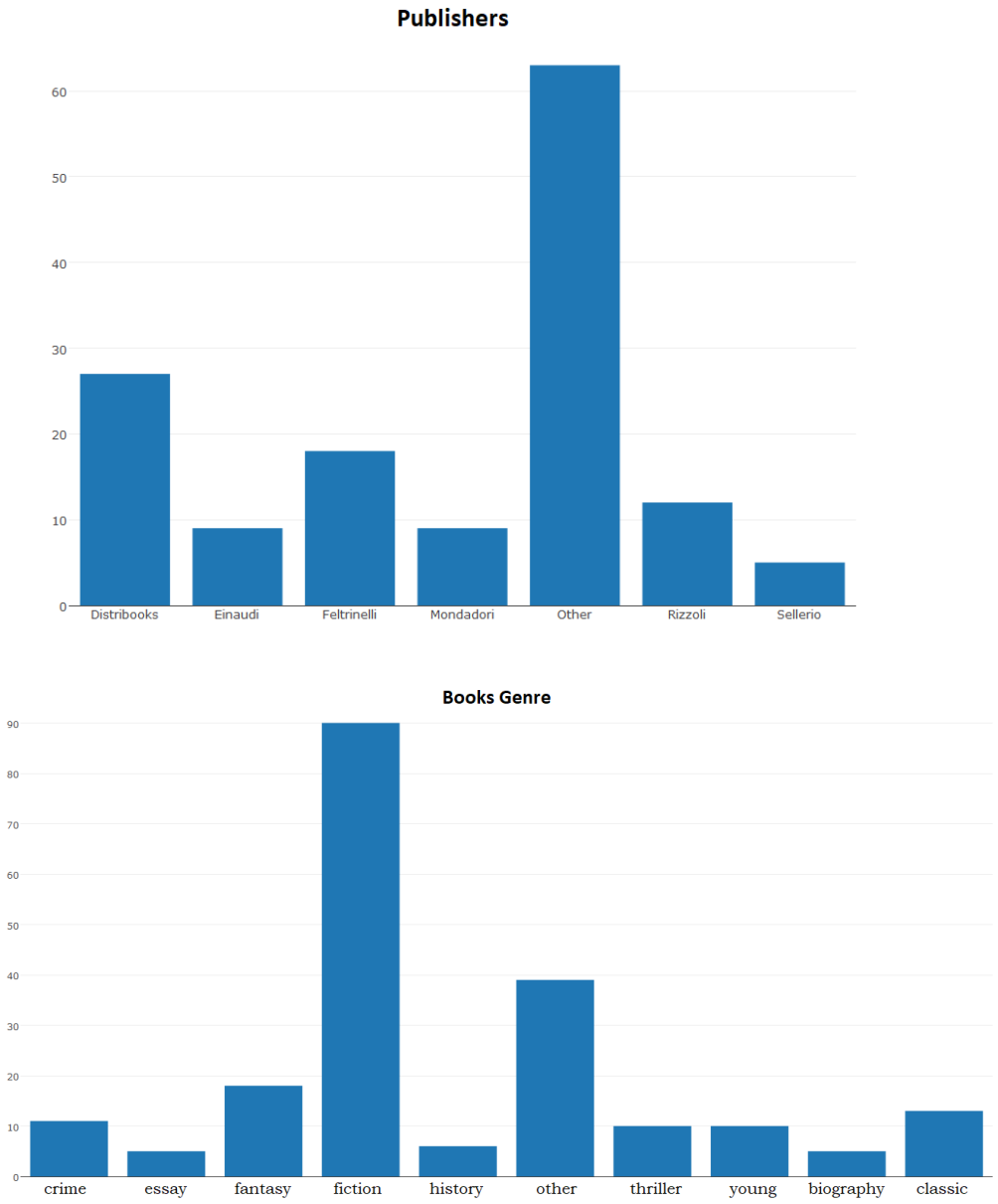


Figure 4: Bar-plot describing the features of books rated by Italian users.

6.2.2 Model

Let us now enter into the details of the model we used for this data.
As previously explained, the objective of the recommender system is to recommend to a user i the item j that he would like with the highest probability. Thus

$$j^* = arg \max_{j \in \{1,2,\cdots,M\}} P(L_{ij} = 1 | \mathbf{f}_j).$$

This probability is computed as

$$\log(odds_{ij}) = \log\Big(\frac{P(L_{ij}=1)}{P(L_{ij}=0)}\Big) + \sum_{k=1}^{K} \log\left(\frac{P(f_{jk}=1|L_{ij}=1)}{P(f_{jk}=1|L_{ij}=0)}\right).$$

Notice that the original ratings r_{ij} in the *Book Crossing* dataset are not binary, as in the model proposed before, but ranged between 0 and 10; thus we have to change it and to opportunely adapt it to our problem, in order to include also this information.

The probability that book j, which user i gave rating $\bar{r}$, has feature k cannot be computed as in Equation 1, but we propose:

$$logit\big(P(f_{jk}=1|L_{ij}=1)\big) = \mu_k^1 + \psi_{ik}^1 + \chi_{ik}^1 \tag{3}$$

and

$$logit\big(P(f_{jk}=1|L_{ij}=0)\big) = \mu_k^0 + \psi_{ik}^0 + \chi_{ik}^0 \tag{4}$$

where

μ_k^1, μ_k^0 is the feature-level random effect,

ψ_{ik}^1, ψ_{ik}^0 is the effect of the user demographics,

χ_{ik}^1 is the effect of liking the items that have feature k

χ_{ik}^0 is the effect of disliking the items that have feature k.

Precisely, the component that includes the demographic information of the user is

$$\psi_{ik}^1 = \beta_{0_k}^1 + \beta_{1_k}^1 x_{i1} + \cdots + \beta_{Q_k}^1 x_{iQ},$$
$$\psi_{ik}^0 = \beta_{0_k}^0 + \beta_{1_k}^0 x_{i1} + \cdots + \beta_{Q_k}^0 x_{iQ}.$$

On the other hand, the relationship between the tastes of the user and the characteristics of the book is kept in χ. In particular,

$$\chi_{ik}^1 = \alpha_k w_{ik}^1,$$

where

$$w_{ik}^1 = \frac{1}{\sum_{j=1}^{M} r_{ij}\big|_{r_{ij}\geq 6}} \mathbf{r}_i\big|_{r_{ij}\geq 6} (F^T)_k,$$

and

$$\chi_{ik}^0 = \alpha_k w_{ik}^0,$$

where

$$w_{ik}^0 = \frac{1}{\sum_{j=1}^{M} r_{ij}\big|_{r_{ij}<6}} \mathbf{r}_i\big|_{r_{ij}<6} (F^T)_k.$$

Notice that w_{ik}^1/ w_{ik}^0 are the sum of the positive/ negative ratings that user i gave to books with feature k, normalized by the sum of all the positive/negative ratings of user i: this is an index of the influence of feature k in the tastes of user i.

The priors that we assign are:

$$\mu_k^1 \sim \mathcal{N}(0, 100000)$$
$$\mu_k^0 \sim \mathcal{N}(0, 100000)$$

$$\beta_{qk}^1 \sim \mathcal{N}(0, \frac{1}{\tau_{qk}})$$
$$\beta_{qk}^0 \sim \mathcal{N}(0, \frac{1}{\tau_{qk}})$$

$$\alpha_k^1 \sim \mathcal{N}(0, \frac{1}{\eta_k})$$
$$\alpha_k^0 \sim \mathcal{N}(0, \frac{1}{\eta_k})$$

$$\tau_{sk} \sim \Gamma(0.001, 0.001)$$
$$\eta_k \sim \Gamma(0.001, 0.001).$$

Moreover, we assign to the probability that user i likes book j a Beta prior:

$$L_{ij}|p \sim \mathcal{B}](p), \quad p \sim Beta(a, b)$$
$$\text{with } a, b \text{ s.t. } \frac{a}{a+b} = \text{empirical mean.}$$

6.2.3 Analysis

We pre-process data in R (version 3.1.3): the *Book Crossing* dataset, as we downloaded it from `http://www2.informatik.uni-freiburg.de/~cziegler/BX/`, is highly noisy and dirty. We remove samples with missing values, clean the incoherence, merge the information obtained through Google Books API, and transform data in the format we described before.
We divide the data into a training set and a testing set: the numbers of users of these datasets is reported in Table 1.

Training data is partitioned into M subsets, transformed in the correct data format calling `rstan_dump` and finally the Markovian subchains from each subset of data is generated thanks to *cmdstan*.
In particular, the *World-wide Book Crossing* dataset is divided into $M = 10$ subsets of 180 users each, and the *Italian Book Crossing* dataset into $M = 3$ subsets of 66 users each.
The sampling of Markovian subchains is done with:

Table 1: Cardinality of training and testing sets.

	Training	**Testing**
World-wide Book Crossing dataset	1800	60
Italian Book Crossing dataset	198	40

total number of iterations = 25000,

burnin = 5000

thinning = 4,

thus we effectively saved 5000 samples for each chain.
We made diagnostic tests in R, in order to check the convergence of the chains: we plotted the trace-plots and the estimated distribution for each parameter.

For each subset m of data, we have a subchain for the parameters associated to each books' feature:

$$\{\beta_k^{m1}\}_{k=1:K}, \quad \{\alpha_k^{m1}\}_{k=1:K}, \quad \{\mu_k^{m1}\}_{k=1:K}, \quad m = 1:M \quad \text{and}$$
$$\{\beta_k^{m0}\}_{k=1:K}, \quad \{\alpha_k^{m0}\}_{k=1:K}, \quad \{\mu_k^{m0}\}_{k=1:K}, \quad m = 1:M.$$

Given those Stan chains, the PART algorithm returns a unique aggregated chain for each parameter: we will refer to these PART chains as $\beta_k^{1*}, \alpha_k^{1*}, \mu_k^{1*}$ and $\beta_k^{0*}, \alpha_k^{0*}, \mu_k^{0*}$.
In particular, we run the PART algorithm with all the combination of settings: OneStage, Pair-wise aggregation, KD-Median, ML Estimation, Smoothing, Non-Smoothing. Notice that we cannot use the Pair-wise setting for the *Italian Book Crossing* dataset, because we have only 3 subgroups of data and the Pair-wise Aggregation requires at least 4 machines.
We will specify at each time the setting we are talking about.

In order to measure the performance of the recommender system we built, and to predict if a book with certain features could be liked by a user, we compute the following prediction.
Thanks to the ergodic theorem, we can approximate

$$P(f_{jk} = 1 | L_{i_{new}j} = 1) = \frac{1}{G} \sum_{g=1}^{G} \frac{exp\{\mu_{k_g}^{1*} + \beta_{k_g}^{1*} \mathbf{x}_{i_{new}} + \alpha_{k_g}^{1*} w_{i_{new}k}^{1}\}}{1 + exp\{\mu_{k_g}^{1*} + \beta_{k_g}^{1*} \mathbf{x}_{i_{new}} + \alpha_{k_g}^{1*} w_{i_{new}k}^{1}\}}$$

$$P(f_{jk} = 1 | L_{i_{new}j} = 0) = \frac{1}{G} \sum_{g=1}^{G} \frac{exp\{\mu_{k_g}^{0*} + \beta_{k_g}^{0*} \mathbf{x}_{i_{new}} + \alpha_{k_g}^{0*} w_{i_{new}k}^{0}\}}{1 + exp\{\mu_{k_g}^{0*} + \beta_{k_g}^{0*} \mathbf{x}_{i_{new}} + \alpha_{k_g}^{0*} w_{i_{new}k}^{0}\}},$$

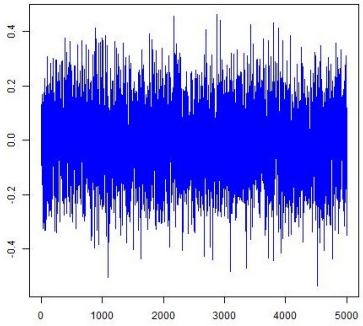
where $\mathbf{w}_{i_{new}}^1, \mathbf{w}_{i_{new}}^0$ are computed excluding all the information about book j.
Then we can compute

$$P(L_{i_{new}j} = 1 | \mathbf{f}_j) = \log\Big(\frac{P(L_{i_{new}j} = 1)}{P(L_{i_{new}j} = 0)}\Big) + \sum_{k=1}^{K} \log\left(\frac{P(f_{jk} = 1 | L_{i_{new}j} = 1)}{P(f_{jk} = 1 | L_{i_{new}j} = 0)}\right). \tag{5}$$

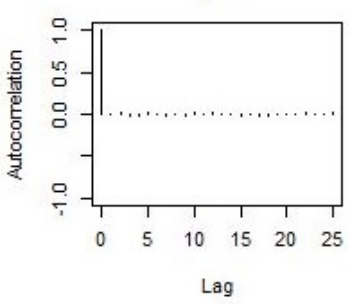
6.2.4 Results

Convergence Diagnostics

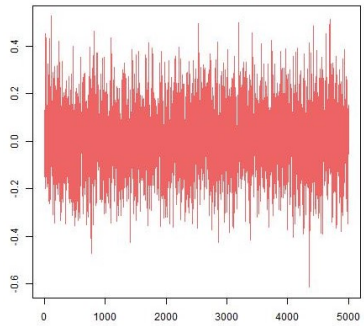
We first check the convergence of the chains obtained with *cmdstan*.
Some of the Monte Carlo chains show a very good behaviour in terms of convergence and precision: for instance, we see the trace-plot and the autocorrelation-plot of the parameter $\alpha_{k_1}^{m_1^1}$ of *Italian Book Crossing* dataset in Figures 5a and 5b, and of parameter $\beta_{2_{k_7}}^{m_{10}^1}$ of *World-wide Book Crossing* dataset in Figures 5a and 5a: the trace-plots are highly dense and autocorrelation is very close to zero, meaning a high mixing and a good convergence.



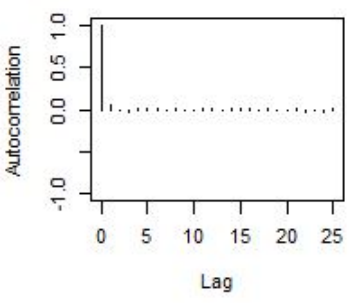
(a) Trace-plot of $\alpha_{k_1}^{m_1^1}$ of *Italian Book Crossing* dataset.



(b) Autocorrelation-plot of $\alpha_{k_1}^{m_1^1}$ of *Italian Book Crossing* dataset



(a) Trace-plot of $\beta_{2_{k_7}}^{m_{10}^1}$ of *World-wide Book Crossing* dataset.



(b) Autocorrelation-plot of $\beta_{2_{k_7}}^{m_{10}^1}$ of *World-wide Book Crossing* dataset

However, since the original full data is partitioned into M subsets, we often see an **information asymmetry** among subgroups: some books' features and users' covariates are rare, then it is highly likely that the information about these features will be poor in some groups.
This asymmetry directly influences the precision of the corresponding Monte Carlo chains. As an example, in Figure 5, 6 and 7, we see the trace-plots of parameter $\beta_{k_{12}}^{m_2^1}$ of the *Italian Book Crossing* dataset in the 3 machines. Feature 12 indicates books about history and the information about them is divided as follows:in subgroup 1 5 users rated books with this feature, in subgroup 2 there

are 2 ratings for history books and in subgroup 3 there are no ratings for books with this feature. This complete lack of information is evident in the traceplot of machine 3.

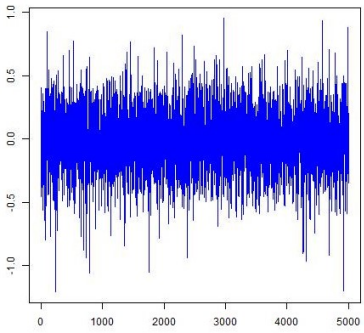


Figure 5: Trace-plot of $\beta_{k\,12}^{m_1^1}$

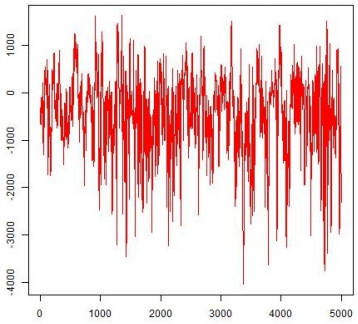


Figure 6: Trace-plot of $\beta_{k\,12}^{m_2^1}$

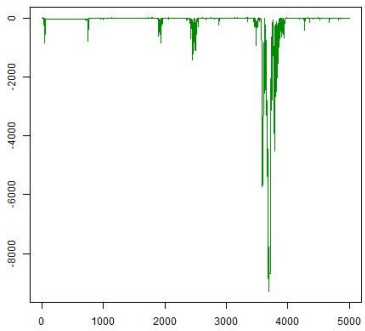


Figure 7: Trace-plot of $\beta_{k\,12}^{m_3^1}$

In order to see what the PART algorithm did, in Figure 9 we can look at the marginal density distribution of the parameters discussed before: we see that the aggregated chains computed by PART seems to be a combination of the single sub-chains.

Figure 8: Marginal density distribution of $\alpha_{k_1}^{m_1^1}$ of *Italian Book Crossing* dataset.

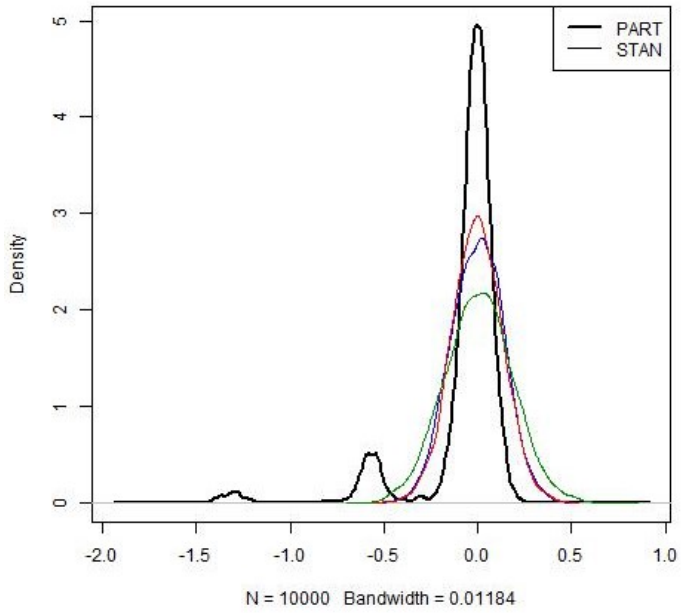
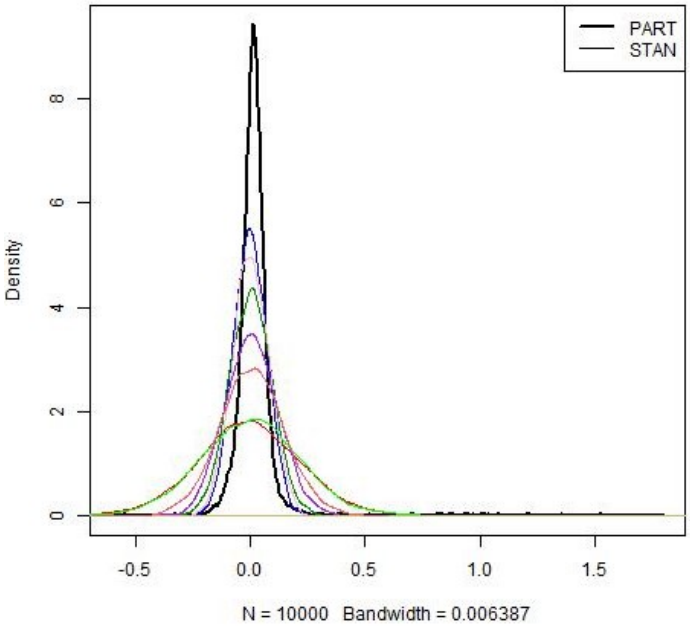


Figure 9: Marginal density distribution of $\beta_{2_{k_7}}^{m_{10}^1}$ of *World-wide Book Crossing* dataset.



Accuracy

Given the chains of the PART algorithm, We measure the goodness of the model by computing the classification accuracy of the system on the testing datasets. The classification accuracy is the percentage of correctly recommended books. For each user i_{new} in the testing set, we check the set of books he rated, and for each j of these books, if the estimated probability $P(L_{i_{new}j} = 1|\mathbf{f}_j)$ is greater or equal to 0.5, we recommend the book. Then we compare the recommendations and the true ratings:

$$\text{accuracy} = \frac{\sum_{i=1}^{N_{test}} 1_{\{\text{recommend}_{ij}=\text{true rating}_{ij}\}}}{\#\text{ratings}}.$$

Let us go into the details of each dataset.

Italian Book Crossing dataset

We consider all the 40 users we kept for testing as people already belonging to the Book Crossing community, thus we have both users' information and some previously rated books. We want to test how good the system we built is by analysing the percentage of right reccomendation it is able to predict.
We compute the probability in 6.2.4 in the four parameters setting of the PART algorithm and we recommend the book if this probability is greater or equal than 0.5. The resulting accuracy are reported in Table 2.

Table 2: *Italian Book Crossing* accuracy

setting	KD OneStage NonSmooth	KD OneStage Smooth	ML OneStage NonSmooth	ML OneStage Smooth
accuracy	28.9%	**59.4%**	375%	**71.9%**

As the analysis of synthetic dataset disclosured, the PART algorithm is more accurate and its estimate more precise when it uses the Maximum Likelihood estimated as cutting point strategy. Moreover, we can say that the smoothing highly influences the result.

World-wide Book Crossing dataset

The *World-wide Book Crossing* testing set is split into two different groups: a set of *new users*, which has just subscribed the community, and a set of *loyal users*, which already rated some books. Thus we compute the predictive in two different ways.

For a *loyal user* i_{loyal}, the probability that he would like a book j that he did not read yet is, as in Equation :

$$P(L_{i_{loyal}j} = 1|\mathbf{f}_j) = \log\Big(\frac{P(L_{i_{loyal}j} = 1)}{P(L_{i_{loyal}j} = 0)}\Big) + \sum_{k=1}^{K} \log\left(\frac{P(f_{jk} = 1|L_{i_{loyal}j} = 1)}{P(f_{jk} = 1|L_{i_{loyal}j} = 0)}\right)$$

with

$$P(f_{jk}=1|L_{i_{loyal}j}=1)=\frac{1}{G}\sum_{g=1}^{G}\frac{exp\{\mu_{k_g}^{1*}+\beta_{k_g}^{1*}\,\mathbf{x}_{i_{loyal}}+\alpha_{k_g}^{1*}\,w_{i_{loyal}k}^{1}\}}{1+exp\{\mu_{k_g}^{1*}+\beta_{k_g}^{1*}\,\mathbf{x}_{i_{loyal}}+\alpha_{k_g}^{1*}\,w_{i_{loyal}k}^{1}\}}$$

$$P(f_{jk}=1|L_{i_{loyal}j}=0)=\frac{1}{G}\sum_{g=1}^{G}\frac{exp\{\mu_{k_g}^{0*}+\beta_{k_g}^{0*}\,\mathbf{x}_{i_{loyal}}+\alpha_{k_g}^{0*}\,w_{i_{loyal}k}^{0}\}}{1+exp\{\mu_{k_g}^{0*}+\beta_{k_g}^{0*}\,\mathbf{x}_{i_{loyal}}+\alpha_{k_g}^{0*}\,w_{i_{loyal}k}^{0}\}},$$

where $\mathbf{w}_{i_{loyal}}^{1}, \mathbf{w}_{i_{loyal}}^{0}$ are computed excluding all the information about book j and retaining the information about the other books that user i_{loyal} rated.

The accuracy of the prediction for *loyal users* is reported in Table 3.

Table 3: *World-wide Book Crossing* accuracy for loyal users.

setting	KD Pariwise NonSmooth	KD Pairwise Smooth	ML Pairwise NonSmooth	ML Pairwise Smooth
accuracy	**63.0%**	18.26%	**68.0%**	**86.3%**

For a *new user* i_{new}, we suppose to have only the information he gives at subscription time: its age and its residency. Thus, the probability that he would like a book j takes into account only the user covariates:

$$P(L_{i_{new}j}=1|\mathbf{f}_j)=\log\Big(\frac{P(L_{i_{new}j}=1)}{P(L_{i_{new}j}=0)}\Big)+\sum_{k=1}^{K}\log\left(\frac{P(f_{jk}=1|L_{i_{new}j}=1)}{P(f_{jk}=1|L_{i_{new}j}=0)}\right)$$

with

$$P(f_{jk}=1|L_{i_{new}j}=1)=\frac{1}{G}\sum_{g=1}^{G}\frac{exp\{\mu_{k_g}^{1*}+\beta_{k_g}^{1*}\,\mathbf{x}_{i_{new}}\}}{1+exp\{\mu_{k_g}^{1*}+\beta_{k_g}^{1*}\,\mathbf{x}_{i_{new}}\}}$$

$$P(f_{jk}=1|L_{i_{new}j}=0)=\frac{1}{G}\sum_{g=1}^{G}\frac{exp\{\mu_{k_g}^{0*}+\beta_{k_g}^{0*}\,\mathbf{x}_{i_{new}}\}}{1+exp\{\mu_{k_g}^{0*}+\beta_{k_g}^{0*}\,\mathbf{x}_{i_{new}}\}}.$$

The accuracy of the prediction for *new users* is reported in Table 4.
We see that on average the prediction accuracy computed on loyal users is higher that the one computed on new users: it suggests us that the contribution of the previous ratings on books with the same characteristics has effectively an impact on the final recommendation.

Table 4: *World-wide Book Crossing* accuracy for new users.

setting	KD Pariwise NonSmooth	KD Pairwise Smooth	ML Pairwise NonSmooth	ML Pairwise Smooth
accuracy	**68.8%**	15.1%	**66.0%**	**71.4%**

A Default Parameters

File of default parameters for PART algorithm.

Listing 17: examples/parameters/PART/defaultParamsPART.par

```
% Number of subsets
M = 4
% Pairwise Aggregation
pairwise_aggregation = 1
% Improve Matching
improve_matching = 1
% Halve
halve = 0
% Number of Trees
ntree = 16
% KD
kd_cut = 1
% Minimum fraction of points per block
min_frac_points_block = 0.01
% min_cut_length
min_cut_length = 0.001
% number of samples to resample
resample_N = 10000
% Gaussian Smooth
gaussian_smooth = 1
% Verbose
verbose = 0
```

File of default parameters for MCMC generation with *CmdStan* (see Listing 18)
Each line represent the following parameters:

- `samplingIter`: number of sampling iterations to be performed for each subchain;
- `burnin`: number of warmup iterations;
- `thin`: thinning parameter;
- `pathR`: path into which all *R* files will be found with respect to `Interface` directory (must be coherent with *R* working directory);
- `pathExeStan`: path of executable *Stan* model with respect to `Interface` directory;
- `useInit`: `true` if initialization of sampler defined in external file.

Listing 18: examples/parameters/MCMC/defaultParamsMCMC.txt

```
2000
1000
```

```
1
Rdir
Rdir/logistic_model
0
```

B Tests - Detailed Results

Table 5: Accuracy ($d = 9$, $N = 5000$)

	M=10		M=20		M=40	
	RMSE	r	RMSE	r	RMSE	r
ML One-stage No Smoothing	0.1370	1.0165	0.3527	1.0089	0.2990	0.9844
ML One-stage Smoothing	0.0725	1.0034	0.0828	**0.9989**	0.1843	0.9984
ML Pairwise No Smoothing	0.1921	1.0191	0.1712	1.0401	0.3121	1.0365
ML Pairwise Smoothing	**0.0634**	**0.9945**	**0.1583**	0.9867	**0.1136**	**0.9886**
KD One-stage No Smoothing	0.2591	1.0084	0.2821	1.0583	0.4493	1.1195
KD One-stage Smoothing	**0.0649**	1.0043	0.1460	0.9914	0.2618	**0.9950**
KD Pairwise No Smoothing	0.1654	1.0319	0.1915	1.0891	0.2140	1.2292
KD Pairwise Smoothing	0.0731	**0.9961**	**0.0880**	**0.9944**	**0.1786**	0.9886

Table 6: Accuracy ($d = 9$, $N = 10000$)

	M=10		M=20		M=40	
	RMSE	r	RMSE	r	RMSE	r
ML One-stage No Smoothing	0.1577	1.0101	0.2159	1.0122	0.2524	1.0047
ML One-stage Smoothing	0.1429	0.9925	**0.1180**	0.9945	0.1468	**0.9978**
ML Pairwise No Smoothing	0.2494	1.0098	0.3549	1.0447	0.7342	1.0757
ML Pairwise Smoothing	**0.0321**	**0.9986**	0.1303	**0.9949**	**0.1312**	0.9956
KD One-stage No Smoothing	0.1559	1.0305	0.3593	1.0586	0.3781	1.1427
KD One-stage Smoothing	0.1392	0.9887	0.1400	**0.9993**	0.2537	0.9813
KD Pairwise No Smoothing	0.1578	1.0392	0.2242	1.0930	0.1639	1.2464
KD Pairwise Smoothing	**0.0588**	**0.9972**	**0.0831**	0.9940	**0.1270**	**0.9908**

Table 7: Accuracy ($d = 9$, $N = 25000$)

	M=10		M=20		M=40	
	RMSE	r	RMSE	r	RMSE	r
ML One-stage No Smoothing	0.1477	1.0169	0.1736	1.0150	0.2008	1.0063
ML One-stage Smoothing	**0.0495**	**0.9998**	0.1295	0.9936	0.2341	**0.9951**
ML Pairwise No Smoothing	0.4828	1.0073	0.2451	1.0300	0.2868	1.0701
ML Pairwise Smoothing	0.0868	0.9923	**0.1235**	**0.9989**	**0.1795**	0.9858
KD One-stage No Smoothing	0.2373	1.0202	0.3310	1.0405	0.4201	1.1572
KD One-stage Smoothing	0.0984	0.9935	0.2537	0.9871	0.2216	**0.9946**
KD Pairwise No Smoothing	0.1531	1.0475	0.2126	1.1039	0.2200	1.2722
KD Pairwise Smoothing	**0.0514**	**0.9956**	**0.0947**	**0.9937**	**0.1082**	0.9918

Table 8: Execution Times ($d = 9$)

[min]	N=5000			N=10000			N=25000		
	M=10	M=20	M=40	M=10	M=20	M=40	M=10	M=20	M=40
ML One-stage No Smoothing	0.54	2.15	8.73	1.12	4.54	21.38	1.12	14.34	59.42
ML One-stage Smoothing	0.53	2.13	8.98	1.13	4.50	21.31	1.13	14.32	58.31
ML Pairwise No Smoothing	0.35	0.76	1.59	0.49	1.10	2.14	0.49	2.72	5.40
ML Pairwise Smoothing	0.37	0.77	1.65	0.50	1.10	2.20	0.50	2.70	5.62
KD One-stage No Smoothing	0.02	0.05	0.12	0.04	0.11	0.27	0.04	0.27	0.67
KD One-stage Smoothing	0.02	0.05	0.13	0.04	0.11	0.26	0.04	0.29	0.66
KD Pairwise No Smoothing	0.05	0.11	0.24	0.07	0.14	0.28	0.07	0.31	0.63
KD Pairwise Smoothing	0.05	0.12	0.27	0.07	0.15	0.31	0.07	0.33	0.65

Table 9: Accuracy ($d = 1$, $N = 5000$)

	M=10		M=20		M=40	
	RMSE	r	RMSE	r	RMSE	r
ML One-stage No Smoothing	0.0018	1.0005	0.0168	0.9956	0.0042	1.0011
ML One-stage Smoothing	0.0010	1.0003	**0.0134**	**0.9965**	**0.0035**	**1.0008**
ML Pairwise No Smoothing	**0.0001**	**1.0001**	0.0257	0.9932	0.0223	1.0416
ML Pairwise Smoothing	0.0041	1.0010	0.0967	1.0264	0.0464	1.0121
KD One-stage No Smoothing	0.0093	1.0025	0.0054	0.9985	**0.0016**	**0.9997**
KD One-stage Smoothing	0.0106	1.0029	**0.0034**	**0.9991**	0.0114	0.9969
KD Pairwise No Smoothing	0.0041	1.0011	0.0087	0.9977	0.0141	1.0038
KD Pairwise Smoothing	**0.0017**	**0.9996**	0.0045	0.9989	0.0235	1.0063

Table 10: Accuracy ($d = 1$, $N = 10000$)

	M=10		M=20		M=40	
	RMSE	r	RMSE	r	RMSE	r
ML One-stage No Smoothing	**0.0006**	**1.0002**	0.0021	1.0006	**0.0156**	1.0043
ML One-stage Smoothing	0.0022	1.0006	**0.0018**	1.0005	0.0168	1.0046
ML Pairwise No Smoothing	0.0041	1.0011	0.0070	**0.9982**	0.0187	1.0053
ML Pairwise Smoothing	0.0213	1.0057	0.0159	0.9958	0.0169	**0.9956**
KD One-stage No Smoothing	0.0022	1.0006	0.0035	1.0010	0.0302	1.0083
KD One-stage Smoothing	0.0034	1.0009	0.0044	1.0012	0.0198	1.0055
KD Pairwise No Smoothing	**0.0004**	**1.0001**	**0.0008**	**0.9998**	**0.0126**	**1.0035**
KD Pairwise Smoothing	0.0056	1.0015	0.0057	1.0016	0.0369	1.0099

Table 11: Accuracy ($d = 1$, $N = 25000$)

	M=10		M=20		M=40	
	RMSE	r	RMSE	r	RMSE	r
ML One-stage No Smoothing	0.0012	1.0003	0.0031	1.0009	**0.0085**	**1.0023**
ML One-stage Smoothing	**0.0006**	1.0001	**0.0022**	**1.0006**	0.0100	1.0027
ML Pairwise No Smoothing	0.0032	1.0009	0.0046	1.0013	0.0994	1.0389
ML Pairwise Smoothing	0.0200	**0.9946**	0.0217	1.0059	0.0598	1.0156
KD One-stage No Smoothing	0.0010	1.0003	0.0035	1.0010	**0.0010**	**1.0003**
KD One-stage Smoothing	0.0031	1.0008	0.0030	1.0008	0.0046	1.0012
KD Pairwise No Smoothing	**0.0001**	**1.0001**	0.0027	1.0008	0.0135	1.0036
KD Pairwise Smoothing	0.0035	1.0009	**0.0010**	**0.9998**	0.0140	1.0037

Table 12: Execution Times ($d = 1$)

[min]	N=5000			N=10000			N=25000		
	M=10	M=20	M=40	M=10	M=20	M=40	M=10	M=20	M=40
ML One-stage No Smoothing	0.46	1.88	8.76	0.90	3.59	16.12	0.90	9.63	40.00
ML One-stage Smoothing	0.47	1.95	9.11	0.90	3.57	16.17	0.90	9.67	40.80
ML Pairwise No Smoothing	0.22	0.50	0.97	0.31	0.66	1.25	0.31	1.73	3.24
ML Pairwise Smoothing	0.10	0.20	0.42	0.15	0.29	0.55	0.15	0.75	1.42
KD One-stage No Smoothing	0.01	0.02	0.05	0.02	0.04	0.08	0.02	0.09	0.20
KD One-stage Smoothing	0.01	0.02	0.05	0.02	0.04	0.08	0.02	0.09	0.20
KD Pairwise No Smoothing	0.02	0.04	0.08	0.02	0.05	0.10	0.02	0.09	0.18
KD Pairwise Smoothing	0.02	0.04	0.08	0.02	0.05	0.10	0.02	0.09	0.18

Table 13: Accuracy ($d = 29$, $N = 5000$)

	M=10		M=20		M=40	
	RMSE	r	RMSE	r	RMSE	r
ML One-stage No Smoothing	0.3847	1.0580	0.5702	1.1253	1.1111	1.4011
ML One-stage Smoothing	**0.1627**	1.0056	**0.2594**	1.0088	**0.3462**	**1.0160**
ML Pairwise No Smoothing	0.2358	1.0765	0.5140	1.2208	0.9102	1.7966
ML Pairwise Smoothing	0.2024	**1.0017**	0.3415	**1.0008**	0.5974	1.0460
KD One-stage No Smoothing	0.3062	1.0733	0.5535	1.2253	1.2981	1.8259
KD One-stage Smoothing	0.1677	1.0074	0.2265	1.0113	0.3824	1.0227
KD Pairwise No Smoothing	0.3023	1.0892	0.5660	1.2645	1.2275	1.9219
KD Pairwise Smoothing	**0.0953**	**1.0028**	**0.1784**	**1.0009**	**0.1504**	**1.0033**

Table 14: Accuracy ($d = 29$, $N = 10000$)

	M=10		M=20		M=40	
	RMSE	r	RMSE	r	RMSE	r
ML One-stage No Smoothing	0.2974	1.0562	0.5978	1.1239	1.0992	1.4249
ML One-stage Smoothing	**0.1513**	1.0048	**0.2414**	1.0088	**0.4266**	**1.0038**
ML Pairwise No Smoothing	0.2096	1.0862	0.4247	1.2486	1.0555	1.7845
ML Pairwise Smoothing	0.1756	**1.0010**	0.3742	**1.0082**	0.6641	1.0324
KD One-stage No Smoothing	0.2936	1.0886	0.6135	1.2399	1.1043	1.8605
KD One-stage Smoothing	0.1425	1.0041	0.2528	1.0037	0.3735	1.0276
KD Pairwise No Smoothing	0.3030	1.0949	0.6169	1.2701	1.2320	1.9608
KD Pairwise Smoothing	**0.0920**	**1.0035**	**0.1236**	**1.0024**	**0.1311**	**1.0066**

Table 15: Accuracy ($d = 29$, $N = 25000$)

	M=10		M=20		M=40	
	RMSE	r	RMSE	r	RMSE	r
ML One-stage No Smoothing	0.2988	1.0523	0.6551	1.1501	1.2532	1.4758
ML One-stage Smoothing	0.1683	1.0047	**0.2648**	**1.0037**	0.2839	1.0142
ML Pairwise No Smoothing	0.2798	1.1000	0.4756	1.2563	1.0672	1.8467
ML Pairwise Smoothing	**0.0935**	**1.0004**	0.2886	1.0176	**0.2170**	**1.0015**
KD One-stage No Smoothing	0.2829	1.0997	0.6211	1.2772	1.2849	1.9106
KD One-stage Smoothing	0.1754	1.0011	0.2498	1.2945	0.2993	1.0266
KD Pairwise No Smoothing	0.2663	1.1031	0.6808	1.0054	1.2147	2.0074
KD Pairwise Smoothing	**0.0886**	**1.0008**	**0.1013**	**0.9991**	**0.1344**	**1.0030**

Table 16: Execution Times ($d = 29$)

[min]	N=5000			N=10000			N=25000		
	M=10	M=20	M=40	M=10	M=20	M=40	M=10	M=20	M=40
ML One-stage No Smoothing	1.12	3.76	14.61	2.44	8.54	43.39	2.44	32.10	134.68
ML One-stage Smoothing	1.11	3.79	14.85	2.37	8.53	42.74	2.37	32.04	133.41
ML Pairwise No Smoothing	0.79	1.71	3.53	1.16	2.48	4.91	1.16	6.38	12.34
ML Pairwise Smoothing	0.83	1.82	3.58	1.22	2.58	5.07	1.22	6.46	13.64
KD One-stage No Smoothing	0.07	0.16	0.34	0.13	0.29	0.69	0.13	0.76	3.09
KD One-stage Smoothing	0.07	0.15	0.34	0.13	0.30	0.72	0.13	0.80	2.53
KD Pairwise No Smoothing	0.16	0.36	0.81	0.22	0.52	1.05	0.22	1.14	2.23
KD Pairwise Smoothing	0.20	0.43	0.92	0.26	0.59	1.18	0.26	1.20	2.44

Table 17: $min\_frac\_points\_block = 0.001$

		ML One-stage Smoothing	ML Pairwise Smoothing	KD One-stage Smoothing	KD Pairwise Smoothing
M=10	RMSE	**0.3938**	0.5249	0.7358	**0.5608**
	r	**1.0210**	1.0298	1.0638	**1.0394**
	Space Partition [ms]	4745	**2706**	**297**	422
	Execution Time [ms]	5061	**3137**	**677**	1043
M=20	RMSE	**0.7611**	0.9715	1.3992	**0.1785**
	r	**0.9690**	1.0489	0.9915	**0.9979**
	Space Partition [ms]	19788	**5639**	**628**	1110
	Execution Time [ms]	20280	**6500**	**1110**	2192
M=40	RMSE	**0.5265**	1.1539	1.3615	**1.1558**
	r	**0.9838**	0.8970	1.0925	**0.9947**
	Space Partition [ms]	84791	**11685**	**1401**	1727
	Execution Time [ms]	85763	**13432**	**2312**	4253

Table 18: $min\_frac\_points\_block = 0.01$

		ML One-stage Smoothing	ML Pairwise Smoothing	KD One-stage Smoothing	KD Pairwise Smoothing
M=10	RMSE	0.4682	**0.1445**	0.2139	**0.0976**
	r	**0.9942**	1.0001	0.9948	**0.9963**
	Space Partition [ms]	2288	**1383**	**69**	171
	Execution Time [ms]	2586	**1731**	**334**	594
M=20	RMSE	0.4862	**0.2630**	0.5037	**0.1508**
	r	1.0112	**0.9965**	0.9794	**0.9953**
	Space Partition [ms]	8423	**2790**	**164**	354
	Execution Time [ms]	8923	**3439**	**632**	1076
M=40	RMSE	0.4482	**0.4235**	0.5657	**0.1201**
	r	**0.9775**	0.9661	1.0286	**0.9971**
	Space Partition [ms]	33312	**5907**	**418**	721
	Execution Time [ms]	34181	**7238**	**1306**	2162

Table 19: Space Partition: $min\_frac\_points\_block = 0.1$, $ntree = 1$

		ML One-stage Smoothing	ML Pairwise Smoothing	KD One-stage Smoothing	KD Pairwise Smoothing
M=10	RMSE	0.2318	**0.0747**	0.2676	**0.0661**
	r	0.9844	**0.9953**	1.0086	**1.0023**
	Space Partition [ms]	1668	**880**	**43**	74
	Execution Time [ms]	1938	**1211**	**310**	443
M=20	RMSE	0.1900	**0.0930**	0.4818	**0.1272**
	r	1.0026	**0.9910**	0.9669	**0.9948**
	Space Partition [ms]	8256	**1919**	**103**	171
	Execution Time [ms]	8767	**2528**	**570**	851
M=40	RMSE	0.5045	**0.1379**	0.6825	**0.2322**
	r	1.0370	**0.9872**	0.9528	**0.9822**
	Space Partition [ms]	31248	**3912**	**264**	362
	Execution Time [ms]	32177	**5139**	**1135**	1618

Table 20: $min\_cut\_length = 0.001$

		ML One-stage Smoothing	ML Pairwise Smoothing	KD One-stage Smoothing	KD Pairwise Smoothing
M=10	RMSE	0.4682	**0.1445**	0.2139	**0.0976**
	r	**0.9942**	1.0001	0.9948	**0.9963**
	Space Partition [ms]	2288	**1383**	**69**	171
	Execution Time [ms]	2586	**1731**	**334**	594
M=20	RMSE	0.4862	**0.2630**	0.5037	**0.1508**
	r	1.0112	**0.9965**	0.9794	**0.9953**
	Space Partition [ms]	8423	**2790**	**164**	354
	Execution Time [ms]	8923	**3439**	**632**	1076
M=40	RMSE	0.4482	**0.4235**	0.5657	**0.1201**
	r	**0.9775**	0.9661	1.0286	**0.9971**
	Space Partition [ms]	33312	**5907**	**418**	721
	Execution Time [ms]	34181	**7238**	**1306**	2162

Table 21: $min\_cut\_length = 0.01$

		ML One-stage Smoothing	ML Pairwise Smoothing	KD One-stage Smoothing	KD Pairwise Smoothing
M=10	RMSE	**0.2564**	0.3951	0.2569	**0.2034**
	r	**0.9801**	0.9793	**0.9895**	0.9813
	Space Partition [ms]	2178	**1237**	**68**	154
	Execution Time [ms]	2455	**1593**	**337**	554
M=20	RMSE	0.3092	**0.2839**	0.6623	**0.2836**
	r	1.0166	**0.9968**	1.0438	**0.9744**
	Space Partition [ms]	8985	**2946**	**170**	348
	Execution Time [ms]	9498	**3621**	**637**	1127
M=40	RMSE	0.8307	**0.2758**	0.6581	**0.1957**
	r	1.0578	**0.9936**	0.9688	**0.9935**
	Space Partition [ms]	36331	**5914**	**432**	751
	Execution Time [ms]	37228	**7243**	**1369**	2199

Table 22: $min\_cut\_length = 0.1$

		ML One-stage Smoothing	ML Pairwise Smoothing	KD One-stage Smoothing	KD Pairwise Smoothing
M=10	RMSE	**0.1831**	0.3362	0.3083	**0.0514**
	r	**0.9984**	0.9731	**0.9936**	1.0048
	Space Partition [ms]	2291	**1260**	**68**	177
	Execution Time [ms]	2564	**1629**	**332**	582
M=20	RMSE	0.6319	**0.1987**	0.5475	**0.1347**
	r	1.0177	**0.9807**	1.0080	**0.9882**
	Space Partition [ms]	8727	**2817**	**164**	373
	Execution Time [ms]	9239	**3518**	**625**	1138
M=40	RMSE	0.4784	**0.3998**	0.5835	**0.2499**
	r	0.9873	**0.9703**	1.0070	**1.0004**
	Space Partition [ms]	38038	**5835**	**423**	746
	Execution Time [ms]	38924	**7125**	**1308**	2165

Table 23: $ntree = 1$

		ML One-stage Smoothing	ML Pairwise Smoothing	KD One-stage Smoothing	KD Pairwise Smoothing
M=10	RMSE	0.4682	**0.1445**	0.2139	**0.0976**
	r	**0.9942**	1.0001	0.9948	**0.9963**
	Space Partition [ms]	2288	**1383**	**69**	171
	Execution Time [ms]	2586	**1731**	**334**	594
M=20	RMSE	0.4862	**0.2630**	0.5037	**0.1508**
	r	1.0112	**0.9965**	0.9794	**0.9953**
	Space Partition [ms]	8423	**2790**	**164**	354
	Execution Time [ms]	8923	**3439**	**632**	1076
M=40	RMSE	0.4482	**0.4235**	0.5657	**0.1201**
	r	**0.9775**	0.9661	1.0286	**0.9971**
	Space Partition [ms]	33312	**5907**	**418**	721
	Execution Time [ms]	34181	**7238**	**1306**	2162

Table 24: $ntree = 16$

		ML One-stage Smoothing	ML Pairwise Smoothing	KD One-stage Smoothing	KD Pairwise Smoothing
M=10	RMSE	0.0503	**0.0812**	0.1463	**0.0492**
	r	**0.9993**	0.9935	0.9900	**0.9996**
	Space Partition [ms]	31785	**21491**	**1097**	2543
	Execution Time [ms]	32062	**22033**	**1372**	3270
M=20	RMSE	0.1629	**0.1189**	0.0959	**0.0783**
	r	**0.9968**	0.9893	1.0048	**0.9944**
	Space Partition [ms]	123908	**44669**	**2618**	5632
	Execution Time [ms]	124458	**45817**	**3092**	7137
M=40	RMSE	**0.2060**	0.2142	0.2215	**0.0969**
	r	**0.9941**	0.9784	0.9909	**0.9935**
	Space Partition [ms]	485227	**95008**	**6769**	12067
	Execution Time [ms]	486173	**97532**	**7668**	15274

Table 25: $ntree = 32$

		ML One-stage Smoothing	ML Pairwise Smoothing	KD One-stage Smoothing	KD Pairwise Smoothing
M=10	RMSE	**0.0504**	0.0575	**0.0381**	0.0623
	r	1.0010	**0.9954**	**0.9986**	0.9978
	Space Partition [ms]	61636	**42550**	**2140**	5133
	Execution Time [ms]	61936	**43288**	**2420**	6134
M=20	RMSE	**0.1134**	0.1902	0.1013	**0.0873**
	r	**0.9972**	0.9858	**0.9978**	0.9925
	Space Partition [ms]	240438	**89246**	**5255**	11342
	Execution Time [ms]	241018	**90805**	**5741**	13445
M=40	RMSE	**0.1048**	0.1960	0.2279	**0.1158**
	r	**0.9936**	0.9803	**0.9928**	0.9911
	Space Partition [ms]	974242	**188902**	**13342**	24982
	Execution Time [ms]	975249	**192038**	**14257**	29459

Table 26: Improved vs not-improved matching

		ML Pairwise Smoothing		KD Pairwise Smoothing	
		Improved	Not Improved	Improved	Not Improved
M=10	RMSE	**0.0634**	0.1567	**0.0731**	0.0747
	r	**0.9945**	0.9929	0.9961	**0.9965**
	Execution Time [ms]	49719	**20724**	11884	**3126**
M=20	RMSE	**0.1583**	0.2226	**0.0880**	0.8000
	r	**0.9867**	0.9819	0.9944	**0.9965**
	Execution Time [ms]	109126	**45782**	25708	**7032**
M=40	RMSE	**0.1136**	0.2167	0.1786	**0.1502**
	r	**0.9886**	0.9847	0.9866	**0.9876**
	Execution Time [ms]	215010	**96052**	55083	**14802**

Table 27: Halving vs not-halving ($min\_frac\_points\_block = 0.001$)

		ML Pairwise Smoothing		KD Pairwise Smoothing	
		Halved	Not Halved	Halved	Not Halved
M=10	RMSE	**0.1747**	0.4486	**0.0655**	0.0735
	r	**0.9885**	0.9616	**0.9956**	1.0007
	Execution Time [ms]	**32743**	45224	**4980**	7928
M=20	RMSE	**0.2018**	0.3933	**0.0448**	0.0808
	r	**0.9907**	1.0174	**0.9989**	1.0040
	Execution Time [ms]	**59714**	94362	**9228**	16974
M=40	RMSE	**0.2357**	0.8876	0.0928	**0.2181**
	r	**0.9769**	1.0372	**0.9942**	0.9904
	Execution Time [ms]	**106862**	197417	**15421**	34608

Table 28: Halving vs not-halving ($min\_frac\_points\_block = 0.01$)

		ML Pairwise Smoothing		KD Pairwise Smoothing	
		Halved	Not Halved	Halved	Not Halved
M=10	RMSE	0.1008	**0.0812**	0.0534	**0.0492**
	r	0.9899	**0.9935**	0.9984	**0.9996**
	Execution Time [ms]	**19624**	22033	**2274**	3270
M=20	RMSE	0.1342	**0.1189**	**0.0736**	0.0783
	r	**0.9961**	0.9893	**0.9949**	0.9944
	Execution Time [ms]	**40043**	45817	**4156**	7137
M=40	RMSE	**0.1191**	0.2142	0.1231	**0.0969**
	r	**0.9906**	0.9784	0.9928	**0.9935**
	Execution Time [ms]	**69710**	97532	**7400**	15274

Table 29: Number of samples from aggregated posterior

	ML Pairwise Smoothing			KD Pairwise Smoothing		
	RMSE	r	Execution Time [ms]	RMSE	r	Execution Time [ms]
1500	0.0532	0.9974	14739	0.0357	**0.9999**	2584
3000	0.0913	1.0007	**14519**	0.0482	0.9991	2551
4500	0.0993	0.9924	14576	**0.0275**	1.0003	**2508**
6000	0.0855	0.9950	15682	0.0582	0.9969	2637
7500	0.0598	0.9950	18205	0.0379	1.0008	2840
9000	0.1004	0.9910	19998	0.0463	0.9989	2987
10500	**0.0454**	**0.9994**	22462	0.0322	1.0007	3171
12000	0.0814	0.9916	24225	0.0472	0.9987	3405
13500	0.0868	1.0008	27082	0.0661	0.9958	3745
15000	0.0694	0.9956	28939	0.0586	0.9966	3753

Figure 10: *gprof*: *KD one-stage with smoothing* ($d = 29$, $N = 5000$, $M = 10$)

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 56.05      1.53     1.53    16368     0.09     0.14  Tree::findCut(RawMCMC const&, Tree::Node const&, double&, unsigned lon
 17.95      2.02     0.49   245744     0.00     0.00  void std::__introselect<__gnu_cxx::__normal_iterator<double*, std::vec
  9.52      2.28     0.26   245744     0.00     0.00  Tree::checkCut(double, arma::Col<double> const&, double, double)
  4.76      2.41     0.13     8192     0.02     0.04  MCestimate::MCestimate(RawMCMC const&, arma::Col<unsigned long long> c
  3.30      2.50     0.09     8192     0.01     0.01  void arma::op_cov::direct_cov<double>(arma::Mat<double>&, arma::Mat<do
  2.56      2.57     0.07       16     4.38    23.27  Tree::build(RawMCMC const&, Tree::Node&, bool, std::vector<MCestimate,
  1.83      2.62     0.05     8192     0.01     0.02  computeMeanCov(arma::Mat<double> const&, arma::Col<unsigned long long>
  1.47      2.66     0.04   292096     0.00     0.00  std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul,
  0.37      2.67     0.01   255744     0.00     0.00  int std::uniform_int_distribution<int>::operator()<std::mersenne_twist
  0.37      2.68     0.01    81920     0.00     0.00  void std::vector<arma::Mat<double>, std::allocator<arma::Mat<double> >
  0.37      2.69     0.01    16352     0.00     0.00  Tree::Node::generateChild(unsigned long long, double, double, bool, ar
  0.37      2.70     0.01     8254     0.00     0.00  MCestimate::MCestimate(MCestimate const&)
  0.37      2.71     0.01     8192     0.00     0.00  void arma::op_mean::apply<arma::Mat<double> >(arma::Mat<arma::Mat<doub
  0.37      2.72     0.01        1    10.00    10.00  RawMCMC::createDefaultArea()
  0.37      2.73     0.01                             bool arma::diskio::load_csv_ascii<double>(arma::Mat<double>&, std::ist
  0.00      2.73     0.00   626752     0.00     0.00  arma::Mat<unsigned long long>::init_warm(unsigned long long, unsigned
  0.00      2.73     0.00   524192     0.00     0.00  arma::Mat<unsigned long long>::steal_mem_col(arma::Mat<unsigned long l
  0.00      2.73     0.00    73837     0.00     0.00  arma::Mat<double>::init_warm(unsigned long long, unsigned long long)

granularity: each sample hit covers 2 byte(s) for 0.37% of 2.73 seconds

index % time    self  children    called     name
                                    16368             Tree::build(RawMCMC const&, Tree::Node&, bool, std::vector<MCestimate, std::alloc
[1]     85.1    1.53    0.79   16368         Tree::findCut(RawMCMC const&, Tree::Node const&, double&, unsigned long long&, arma::
                0.49    0.00  245744/245744       void std::__introselect<__gnu_cxx::__normal_iterator<double*, std::vector<double,
                0.26    0.00  245744/245744       Tree::checkCut(double, arma::Col<double> const&, double, double) [6]
                0.01    0.03  245744/255744       int std::uniform_int_distribution<int>::operator()<std::mersenne_twister_engine<u
                0.00    0.00   16368/24577        RawMCMC::get_samples() const [32]
                0.00    0.00   16368/73837        arma::Mat<double>::init_warm(unsigned long long, unsigned long long) [31]
                                    16368             aggregatedResampling(Parameters const&, std::vector<arma::Mat<double>, std::alloc
-----------------------------------------------
                0.49    0.00  245744/245744       Tree::findCut(RawMCMC const&, Tree::Node const&, double&, unsigned long long&, ar
[3]     17.9    0.49    0.00  245744         void std::__introselect<__gnu_cxx::__normal_iterator<double*, std::vector<double, std
                0.00    0.00     192/192          void std::__adjust_heap<__gnu_cxx::__normal_iterator<double*, std::vector<double,
-----------------------------------------------
                                    16352             Tree::build(RawMCMC const&, Tree::Node&, bool, std::vector<MCestimate, std::alloc
                                       16             Tree::grow(RawMCMC const&, arma::Col<unsigned long long> const&, bool, std::vectc
[4]     13.6    0.07    0.30      16+16352      Tree::build(RawMCMC const&, Tree::Node&, bool, std::vector<MCestimate, std::allocator
                0.13    0.16    8032/8192         MCestimate::MCestimate(RawMCMC const&, arma::Col<unsigned long long> const&, arma
                0.01    0.00   16352/16352        Tree::Node::generateChild(unsigned long long, double, double, bool, arma::Col<uns
                0.00    0.01     160/160          void std::vector<MCestimate, std::allocator<MCestimate> >::_M_emplace_back_aux<Ra
                0.00    0.00   16352/292096       std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483615u
                0.00    0.00   16352/524192       arma::Mat<unsigned long long>::steal_mem_col(arma::Mat<unsigned long long>&, unsi
                0.00    0.00   12276/626752       arma::Mat<unsigned long long>::init_warm(unsigned long long, unsigned long long)
                                    16368             Tree::findCut(RawMCMC const&, Tree::Node const&, double&, unsigned long long&, ar
                                     8176             aggregatedResampling(Parameters const&, std::vector<arma::Mat<double>, std::alloc
                                    16352             Tree::build(RawMCMC const&, Tree::Node&, bool, std::vector<MCestimate, std::alloc
-----------------------------------------------
                0.00    0.00     160/8192         void std::vector<MCestimate, std::allocator<MCestimate> >::_M_emplace_back_aux<Ra
                0.13    0.16    8032/8192         Tree::build(RawMCMC const&, Tree::Node&, bool, std::vector<MCestimate, std::alloc
[5]     10.6    0.13    0.16    8192          MCestimate::MCestimate(RawMCMC const&, arma::Col<unsigned long long> const&, arma::Ma
                0.05    0.11    8192/8192         computeMeanCov(arma::Mat<double> const&, arma::Col<unsigned long long> const&, un
                0.00    0.00   16384/73837        arma::Mat<double>::init_warm(unsigned long long, unsigned long long) [31]
                0.00    0.00    8192/8208         RawMCMC::get_n_subsets() const [34]
```

Figure 11: *gprof*: *ML one-stage with smoothing* ($d = 29$, $N = 5000$, $M = 10$)

```
Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 41.68      4.83     4.83   331816     0.01     0.02  Tree::ml_cutting_rule(RawMCMC const&, Tree::Node const&, unsigned long long)
 25.45      7.78     2.95    19043     0.15     0.57  Tree::findCut(RawMCMC const&, Tree::Node const&, double&, unsigned long long&
 14.06      9.41     1.63   331816     0.00     0.00  void std::__introsort_loop<__gnu_cxx::__normal_iterator<arma::arma_sort_index
  6.04     10.11     0.70   331816     0.00     0.01  bool arma::arma_sort_index_helper<arma::Mat<double>, false>(arma::Mat<unsigne
  2.93     10.45     0.34     9596     0.04     0.06  MCestimate::MCestimate(RawMCMC const&, arma::Col<unsigned long long> const&,
  2.16     10.70     0.25   663632     0.00     0.00  arma::subview_elem1<unsigned long long, arma::Mat<unsigned long long> >::extr
  1.64     10.89     0.19   331816     0.00     0.00  Tree::checkCut(double, arma::Col<double> const&, double, double)
  0.86     10.99     0.10  7188002     0.00     0.00  arma::Mat<unsigned long long>::init_warm(unsigned long long, unsigned long lo
  0.86     11.09     0.10       16     6.25    42.82  Tree::build(RawMCMC const&, Tree::Node&, bool, std::vector<MCestimate, std::a
  0.86     11.19     0.10     9596     0.01     0.01  void arma::op_cov::direct_cov<double>(arma::Mat<double>&, arma::Mat<double> c
  0.69     11.27     0.08     9596     0.01     0.02  computeMeanCov(arma::Mat<double> const&, arma::Col<unsigned long long> const&
  0.52     11.33     0.06   663648     0.00     0.00  Tree::normalize(std::vector<MCestimate, std::allocator<MCestimate> >&)
  0.43     11.38     0.05  4681786     0.00     0.00  arma::Mat<unsigned long long>::steal_mem_col(arma::Mat<unsigned long long>&,
  0.43     11.43     0.05   380976     0.00     0.00  std::mersenne_twister_engine<unsigned long, 32ul, 624ul, 397ul, 31ul, 2567483
  0.35     11.47     0.04    19160     0.00     0.00  Tree::Node::generateChild(unsigned long long, double, double, bool, arma::Col
  0.26     11.50     0.03   331816     0.00     0.00  void std::__insertion_sort<__gnu_cxx::__normal_iterator<arma::arma_sort_index
  0.17     11.52     0.02                              bool arma::diskio::load_csv_ascii<double>(arma::Mat<double>&, std::istream&,
  0.09     11.53     0.01    95960     0.00     0.00  void std::vector<arma::Mat<double>, std::allocator<arma::Mat<double> > >::_M_
  0.09     11.54     0.01     9596     0.00     0.00  void arma::op_mean::apply<arma::Mat<double> >(arma::Mat<arma::Mat<double>::el
  0.09     11.55     0.01       17     0.59     0.82  posteriorResampling(std::vector<std::vector<MCestimate, std::allocator<MCesti
  0.09     11.56     0.01        1    10.00    10.00  RawMCMC::createDefaultArea()
  0.09     11.57     0.01        1    10.00    20.00  RawMCMC::RawMCMC(std::vector<arma::Mat<double> const*, std::allocator<arma::M
  0.09     11.58     0.01                              arma::unwrap_check_mixed<arma::Mat<double> >::~unwrap_check_mixed()
  0.09     11.59     0.01                              arma::subview<unsigned long long>::extract(arma::Mat<unsigned long long>&, ar
  0.00     11.59     0.00  1745472     0.00     0.00  arma::Mat<double>::init_warm(unsigned long long, unsigned long long)
  0.00     11.59     0.00   360472     0.00     0.00  RawMCMC::get_samples() const
  0.00     11.59     0.00   341816     0.00     0.00  int std::uniform_int_distribution<int>::operator()<std::mersenne_twister_engi
  0.00     11.59     0.00   341428     0.00     0.00  RawMCMC::get_n_subsets() const
  0.00     11.59     0.00   341412     0.00     0.00  RawMCMC::get_mark() const
  0.00     11.59     0.00   331816     0.00     0.00  arma::Mat<double>::operator=(arma::subview<double> const&)
granularity: each sample hit covers 2 byte(s) for 0.09% of 11.59 seconds

index % time    self  children    called     name
                                    19043             Tree::build(RawMCMC const&, Tree::Node&, bool, std::vector<MCestimate, std::allocator<
[1]     93.5    2.95    7.88   19043         Tree::findCut(RawMCMC const&, Tree::Node const&, double&, unsigned long long&, arma::Col<d
                4.83    2.80  331816/331816      Tree::ml_cutting_rule(RawMCMC const&, Tree::Node const&, unsigned long long) [3]
                0.19    0.02  331816/331816      Tree::checkCut(double, arma::Col<double> const&, double, double) [9]
                0.00    0.04  331816/341816      int std::uniform_int_distribution<int>::operator()<std::mersenne_twister_engine<unsign
                0.00    0.00   19043/360472      RawMCMC::get_samples() const [39]
                0.00    0.00   19043/1745472     arma::Mat<double>::init_warm(unsigned long long, unsigned long long) [38]
                                    19043             aggregatedResampling(Parameters const&, std::vector<arma::Mat<double>, std::allocator<
-----------------------------------------------
                4.83    2.80  331816/331816      Tree::findCut(RawMCMC const&, Tree::Node const&, double&, unsigned long long&, arma::C
[3]     65.8    4.83    2.80  331816        Tree::ml_cutting_rule(RawMCMC const&, Tree::Node const&, unsigned long long) [3]
                0.70    1.66  331816/331816      bool arma::arma_sort_index_helper<arma::Mat<double>, false>(arma::Mat<unsigned long lo
                0.25    0.00  663632/663632      arma::subview_elem1<unsigned long long, arma::Mat<unsigned long long> >::extract(arma:
                0.04    0.05 3318160/4681786     arma::Mat<unsigned long long>::steal_mem_col(arma::Mat<unsigned long long>&, unsigned
                0.06    0.02  663632/663648      Tree::normalize(std::vector<MCestimate, std::allocator<MCestimate> >&) [14]
                0.03    0.00 2386139/7188002     arma::Mat<unsigned long long>::init_warm(unsigned long long, unsigned long long) [12]
                0.00    0.00 1659080/1745472     arma::Mat<double>::init_warm(unsigned long long, unsigned long long) [38]
                0.00    0.00  331816/341428      RawMCMC::get_n_subsets() const [40]
                0.00    0.00  331816/360472      RawMCMC::get_samples() const [39]
                0.00    0.00  331816/331816      arma::Mat<double>::operator=(arma::subview<double> const&) [42]
                0.00    0.00  331816/341412      RawMCMC::get_mark() const [41]
-----------------------------------------------
                0.70    1.66  331816/331816      Tree::ml_cutting_rule(RawMCMC const&, Tree::Node const&, unsigned long long) [3]
[4]     20.4    0.70    1.66  331816        bool arma::arma_sort_index_helper<arma::Mat<double>, false>(arma::Mat<unsigned long long>&
                1.63    0.00  331816/331816      void std::__introsort_loop<__gnu_cxx::__normal_iterator<arma::arma_sort_index_packet<d
                0.03    0.00  331816/331816      void std::__insertion_sort<__gnu_cxx::__normal_iterator<arma::arma_sort_index_packet<d
-----------------------------------------------
                                  5289026             void std::__introsort_loop<__gnu_cxx::__normal_iterator<arma::arma_sort_index_packet<d
                1.63    0.00  331816/331816      bool arma::arma_sort_index_helper<arma::Mat<double>, false>(arma::Mat<unsigned long lo
[5]     14.1    1.63    0.00  331816+5289026 void std::__introsort_loop<__gnu_cxx::__normal_iterator<arma::arma_sort_index_packet<doubl
                0.00    0.00      62/62          void std::__adjust_heap<__gnu_cxx::__normal_iterator<arma::arma_sort_index_packet<doub
                                  5289026             void std::__introsort_loop<__gnu_cxx::__normal_iterator<arma::arma_sort_index_packet<d
```

Figure 12: *Valgrind*: *KD pairwise with smoothing* ($d = 9$, $N = 25000$, $M = 40$)

```
==5321== Memcheck, a memory error detector
==5321== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==5321== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==5321== Command: ./main data/input_M40_N5.txt data/outPART_M40_N5_kdPairSmooth.csv parameters/kdPairSmooth.par
==5321== Parent PID: 5125
==5321==
==5321==
==5321== HEAP SUMMARY:
==5321==     in use at exit: 72,704 bytes in 1 blocks
==5321==   total heap usage: 7,770,640 allocs, 7,770,639 frees, 14,568,147,510 bytes allocated
==5321==
==5321== 72,704 bytes in 1 blocks are still reachable in loss record 1 of 1
==5321==    at 0x4C29B1F: malloc (in /u/sw/pkgs/toolchains/gcc-glibc/5/base/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==5321==    by 0x7A20F7F: _GLOBAL__sub_I_eh_alloc.cc (in /u/sw/pkgs/toolchains/gcc-glibc/5/prefix/lib/libstdc++.so.6.0.21)
==5321==    by 0x400F289: call_init.part.0 (in /u/sw/pkgs/toolchains/gcc-glibc/5/prefix/lib/ld-2.23.so)
==5321==    by 0x400F39A: _dl_init (in /u/sw/pkgs/toolchains/gcc-glibc/5/prefix/lib/ld-2.23.so)
==5321==    by 0x4000C99: ??? (in /u/sw/pkgs/toolchains/gcc-glibc/5/prefix/lib/ld-2.23.so)
==5321==    by 0x3: ???
==5321==    by 0xFFEFFF13A: ???
==5321==    by 0xFFEFFF141: ???
==5321==    by 0xFFEFFF157: ???
==5321==    by 0xFFEFFF17C: ???
==5321==
==5321== LEAK SUMMARY:
==5321==    definitely lost: 0 bytes in 0 blocks
==5321==    indirectly lost: 0 bytes in 0 blocks
==5321==      possibly lost: 0 bytes in 0 blocks
==5321==    still reachable: 72,704 bytes in 1 blocks
==5321==         suppressed: 0 bytes in 0 blocks
==5321==
==5321== For counts of detected and suppressed errors, rerun with: -v
==5321== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

C CmdStan

CmdStan is the shell interface of Stan.
We used version `v2.16.0`, already available in `Interface/cmdstan-2.16.0`.

In order to install *CmdStan*, the user should:

- download *CmdStan* from `http://mc-stan.org/users/interfaces/cmdstan`;
- change directories to the *CmdStan* directory;
- build *CmdStan*, using the `-jN` option specifies the number N of CPU cores

```
> make build -j4
```

The model used for synthetic analysis, written in *Stan* language, is reported in Figure 13.

Figure 13: /Interface/Rdir/logistic_model.stan

```
////////////////////////// DATA /////////////////////////////////////
  data {
    int<lower = 0> n;                 // number of data
    int<lower = 0> d;                 // number of coefficients of regression
    int<lower = 0, upper = 1> Y[n];   // response vector
    matrix[d, n] X;                   // design matrix

  }

//////////////////// PARAMETERS ////////////////////////////////
  parameters {
    row_vector[d] beta;        // coefficients of regression
  }


////////////////// MODEL ///////////////////////
  model {

  // Likelihood
    for (s in 1:n)
    {
      Y[s] ~ bernoulli(inv_logit(-3 + beta*X[,s]));
    }

    // Prior
    // beta
    for (j in 1:(d))
    {
      beta[j] ~ normal(0, 5);
    }

  }
```

CmdStan requires data and parameters initialization to have extension *.data.R* and *.init.R* respectively.

Because of that, we use *RStan* function `stan_rdump` to transform necessary files in compatible format:

```
stan_rdump(list, file = "")
```

where `list` is the vector of the names of the R objects to be dumped, and `file` is the character string naming the file where to save the output.
After transformation, the data format is coded as follows:

```
numbers:  Q <- 8
vectors:  Y <- c(0,0,0,0,1,0,1,0,1,0)
matrices: L <- structure(c(1,0,0,0,0,0,0,1), .Dim=c(2,4))
```

The code for format transformation is reported in Figure 14 for data input and in Figure 15 for parameters initialization: their outputs are the files *subdata.data.R* and *inits.init.R*, temporarily saved in the `Rdir` directory.

Once data and initialization have been written in compatible format with *CmdStan*, it is possible to generate MCMC samples executing the *Stan* model. In order to do that, it is necessary to:

- compile the model to generate corresponding *.hpp* and executable.
  To compile the model for synthetic data (`Interface/Rdir/logistic_model.stan`) from *CmdStan* directory:

  ```
  > make ../Rdir/logistic_model
  ```

- run the model from directory in which *.stan* model is located, specifying the parameters for sampling (`sample` method), which are `num_samples` (sampling iterations), `num_warmup` (burnin or warump iterations) and `thin` (thinning).
  To run the model for synthetic tests with *Stan* default parameters, from `Interface/Rdir` type:

  ```
  > ./logistic_model sample
  num_samples=20000 num_warmup=5000 thin=4
  data file=subdata.data.R init=inits.init.R
  output file=output.csv
  ```

The output file *output.csv* is a *Stan* object that can be easily read in R using the *RStan* function `read_stan_csv`.

For more details about *CmdStan*, the manual is available here `https://github.com/stan-dev/cmdstan/releases/download/v2.16.0/cmdstan-guide-2.16.0.pdf`.

Figure 14: /Interface/Rdir/dataDump.R

```r
##### Modify the following path according to your system!!
setwd('PART_BayesPACS-master/Interface/Rdir')

library(MASS)
library(rstan)

subX <- as.matrix(read.table("subdata.csv", header = F))

# number of data
n <- dim(subX)[1]

# number of covariates
d <- dim(subX)[2]

# response vector
b0 <- -3
b  <-  rnorm(d, 0, 25)

odd  <- exp(cbind(b0*rep(1, n),b*subX))
prob <- odd/(odd +1)

Y <- NULL
for (j in seq(n)){
  Y[j] <- rbinom(1,1,prob[j])
}

# design matrix
X <- t(subX)

stan_rdump(c('n','d', 'Y','X'), "subdata.data.R")
```

Figure 15: /Interface/Rdir/initDump.R

```r
##### Modify the following path according to your system!!
setwd('PART_BayesPACS-master/Interface/Rdir')

library(MASS)
library(rstan)

subX <- as.matrix(read.table("subdata.csv", header = F))
d    <- dim(subX)[2]
beta <- rep(0.1,d)

stan_rdump(c('beta'), "inits.init.R")
```

D Google Books API

Here we report the code used to retrieve additional information from Google Books.
In this example we are looking for title, author, genre, number of pages and language of book whose ISBN-13 is 9780971880108.

```
<html>
  <head>
        <title>Books API</title>
  </head>
  <body>
        <div id="content"></div>
        <script>
          function handleResponse(response) {
          for (var i = 0; i < response.items.length; i++) {
                var item = response.items[i];
                document.getElementById("content").innerHTML += "<br>" +
                item.volumeInfo.title + ";" +
                item.volumeInfo.authors + ";" +
                item.volumeInfo.mainCategory  + ";" +
                item.volumeInfo.pageCount + ";" +
                item.volumeInfo.language;
          }
        }
        </script>

        <script
        src="https://www.googleapis.com/books/v1/
volumes?q=isbn:9780971880108&callback=handleResponse">
        </script>

 </body>
</html>
```

E Features of the *Book Crossing* dataset

World-wide Book Crossing dataset

The indicators of world-wide users's covariates are:

x_{i1} indicates if user i is an adolescent (less than 19 years old)

x_{i2} indicates if user i is a young adult (19-34 years old)

x_{i3} indicates if user i is an adult (35-45 years old)

x_{i4} indicates if user i is middle aged (46-60 years old)

x_{i5} indicates if user i lives in Europe

x_{i6} indicates if user i lives in North America.

The indicators of the features of books read by world-wide users are:

f_{j1} indicates if book j is huge (less than 100 pages)

f_{j2} indicates if book j is normal (101-400 pages)

f_{j3} indicates if book j is small (401-700 pages)

f_{j4} indicates if book j is very huge (more than 701 pages)

f_{j5} indicates if book j is about arts

f_{j6} indicates if book j is a biography or autobiography

f_{j7} indicates if book j is a classic

f_{j8} indicates if book j is a comedy

f_{j9} indicates if book j is a crime book

f_{j10} indicates if book j is a drama book

f_{j11} indicates if book j is a fantasy book

f_{j12} indicates if book j is a fiction book

f_{j13} indicates if book j is a history book

f_{j14} indicates if book j is a another genre

f_{j15} indicates if book j is a science fiction

f_{j16} indicates if book j is about social and politics

f_{j17} indicates if book j is a thriller

f_{j18} indicates if book j is for women

f_{j18} indicates if book j is for young people.

Italian Book Crossing dataset

The indicators of Italian users's covariates are:

x_{i1} indicates if user i is an adolescent (less than 19 years old)

x_{i2} indicates if user i is a young adult (19-34 years old)

x_{i3} indicates if user i is an adult (35-45 years old)

x_{i4} indicates if user i is middle aged (46-60 years old)

x_{i5} indicates if user i lives in an administrative center

x_{i6} indicates if user i lives in Sud Italy

x_{i7} indicates if user i lives in Nord Italy

x_{i8} indicates if user i lives in Middle Italy.

The indicators of the features of books rated by Italian users are:

f_{j1} indicates if book j is published by Distribooks

f_{j2} indicates if book j is published by Einaudi

f_{j3} indicates if book j is published by Feltrinelli

f_{j4} indicates if book j is published by Mondadori

f_{j5} indicates if book j is published by others minor editors

f_{j6} indicates if book j is published by Rizzoli

f_{j7} indicates if book j is published by Sellerio

f_{j8} indicates if book j is a crime book

f_{j9} indicates if book j is an essay

f_{j10} indicates if book j is a fantasy book

f_{j11} indicates if book j is a fiction book

f_{j12} indicates if book j is a history book

f_{j13} indicates if book j is a another genre

f_{j14} indicates if book j is a thriller

f_{j15} indicates if book j is for young people.

f_{j16} indicates if book j is a biography or autobiography

f_{j17} indicates if book j is a classic.

References

Condliff, M. K., Lewis, D. D., Madigan, D., and Posse, C. (1999). Bayesian mixed-effects models for recommender systems.

Scott, S. L., Blocker, A. W., Bonassi, F. V., Chipman, H. A., George, E. I., and McCulloch, R. E. (2016). Bayes and big data: The consensus monte carlo algorithm. *International Journal of Management Science and Engineering Management*, 11(2):78–88.

Srivastava, S., Cevher, V., Dinh, Q., and Dunson, D. (2015). Wasp: Scalable bayes via barycenters of subset posteriors. In *Artificial Intelligence and Statistics*, pages 912–920.

Wang, X., Guo, F., Heller, K. A., and Dunson, D. B. (2015). Parallelizing mcmc with random partition trees. In *Advances in Neural Information Processing Systems*, pages 451–459.

Ziegler, C.-N., McNee, S. M., Konstan, J. A., and Lausen, G. (2005). Improving recommendation lists through topic diversification. In *Proceedings of the 14th international conference on World Wide Web*, pages 22–32. ACM.