Complete Package Documentation

Stephen Smith

November 14, 2022

Contents

1 Introduction

In this document, we are going to trace the implementation of the local FCI and local PC algorithms, going into detail about some of the design decisions that were made and for the purposes of determining the accuracy of the implementation.

This package implements both the sample and population versions of the algorithms we are covering. The population versions of the algorithm use the d-separation criterion (implemented by `bnlearn`) to determine conditional independence (CI) relationships (since the population version assumes a CI oracle). The population version of the algorithm also uses the true DAG to identify neighborhoods, while the sample version must estimate these neighborhoods using data. It is possible to provide the true DAG as a Markov Blanket (MB) oracle while using the data for conditional independence inference, but this scenario is primarily for testing purposes and has no real-world relevance.

In this package, most classes and functions contain a boolean variable named `verbose`, which is used to provide debugging or function progress information.

In this guide, I will occasionally refer to so-called "wrapper" functions, a term that designates a function which performs basic tasks (e.g. ensuring data inputs are the correct data type) while calling other functions to handle the actual implementation of the various algorithms. This decision was made to improve code readability and, more importantly, facilitates the integration of functions written in R and in Rcpp.

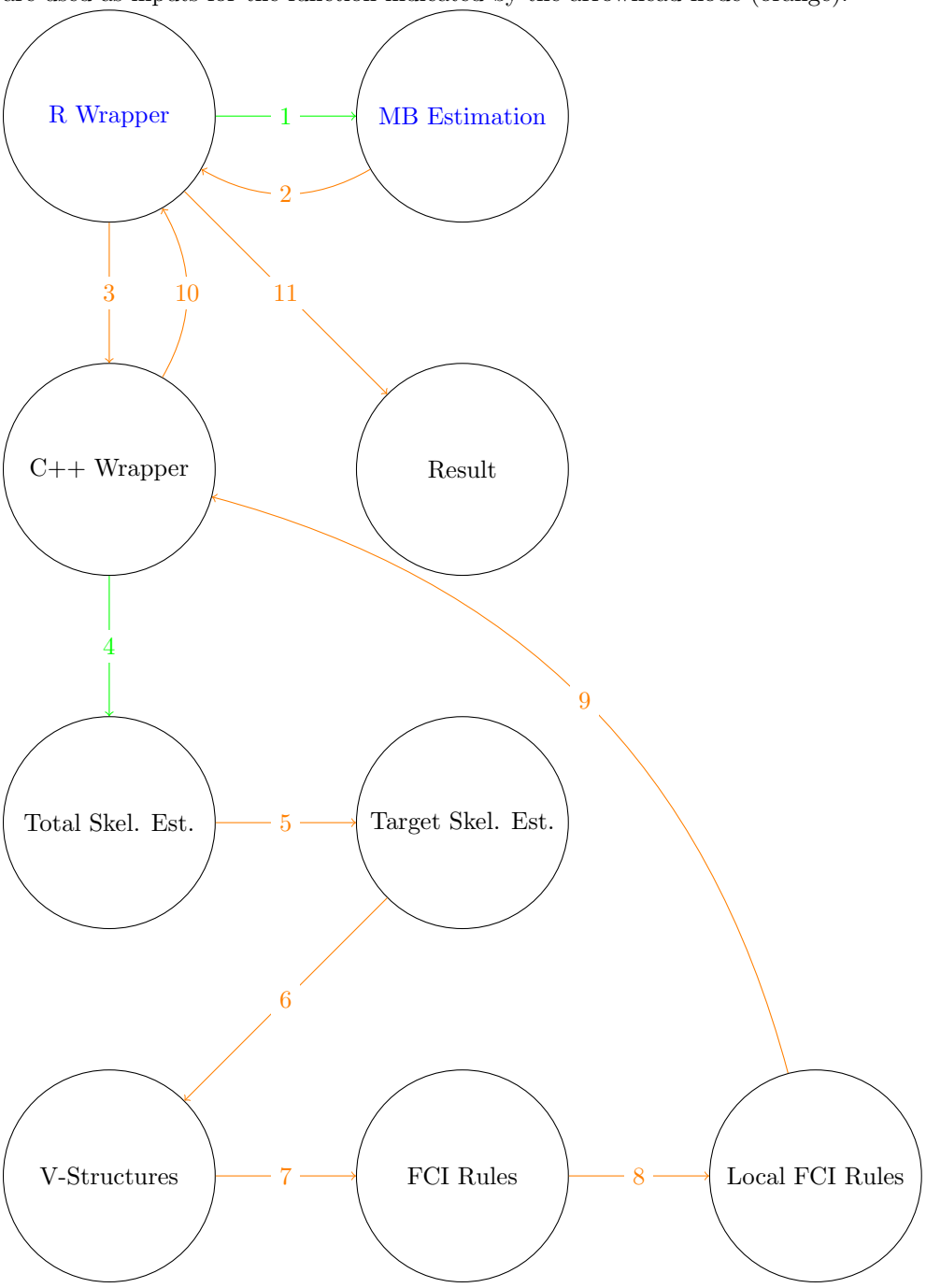
The purpose of this guide is to give a detailed overview of how this package is implemented and tested, ensuring that the code is efficient and correct.

2 Analytical Outline

1. R Wrapper Function
    (a) MB Estimation
        i. MMPC
        ii. SES
        iii. PC-Simple
2. C++ Wrapper

3 Local FCI Function Schematic

This page provides a graphical overview of functions used to implement the local FCI algorithm. Each arrow represents either a sub-step (green) or a progression in the algorithm where the results from the tail node are used as inputs for the function indicated by the arrowhead node (orange).



4 R Wrapper Function

Main Tasks

- Ensure that the data types are appropriate and the inputs are valid
- Provides a function that allows us to both estimate Markov Blankets with R libraries and use Rcpp code for the implementation of hte algorithm
- Carries out all necessary algorithm steps with modularity maintained

This function includes calls to `colMeans` and `stats::cov` in order to store information about our dataset. In particular, we wish to observe if there are significant scaling differences between our variables in the original dataset before using the `scale` function to standardize all of our variables. Next, if the true DAG is given as an input, then we are considering the version of the algorithm where the Markov Blankets are known *a priori*. If not, we will estimate the Markov Blankets for the target nodes and their first-order neighbors using the `MXM` package. We first call `getAllMBs` to obtain all of the stated Markov Blankets, followed by conveniently storing this information in a matrix using the function `getEstInitialDAG`. This function is a bit of a misnomer, since we are not really approximating the true DAG, but we name it thus because we are attempting to approximate what the true DAG provides us, namely information regarding the Markov Blankets of the nodes we are interested in. This is a matrix where the entries of row i indicate which nodes are included in the estimated neighborhood of i. This matrix will be converted in our R implementation using the `MBList` class.

After obtaining the Markov Blankets and ensuring every variable is the correct class, we call the population or sample version of the algorithm implemented in Rcpp. The function returns:

- `amat` - the estimated adjacency matrix produced by the algorithm
- `S` - the separating sets identified by the algorithm
- `NumTests` - the number of conditional independence tests used to identify Markov Blankets and the estimated graph
- `MBNumTests` - the number of conditional independence tests used to identify the Markov Blankets
- `RulesUsed` - a vector providing the number of times each FCI rule was used during edge orientation in the algorithm
- `Nodes` - all the nodes considered by the algorithm, including both target nodes and their first-order neighbors
- `totalSkeletonTime` - the time taken for the first stage in the skeleton estimation process
- `targetSkeletonTimes` - a series of values separated by commas in a string providing the time taken for each target skeleton in the second stage of the skeleton estimation process
- `totalTime` - the total amount of time taken to complete the Rcpp function
- `referenceDAG` - either the true DAG or the matrix providing information about the estimated Markov Blankets
- `mbList` - a list containing the estimated Markov Blankets
- `data_means` - the mean calculated for each variable in the dataset
- `data_cov` - the variance-covariance matrix for the inputted dataset

5 Markov Blanket Estimation Functions

The `MXM` package has a variety of algorithms developed for the general problem of feature selection, including algorithms specifically designed for Markov Blanket estimation.

5.1 Algorithms

5.1.1 MMPC

Algorithm 1 MaxMin Heuristic

Input: Data on variables V ($|V| = p$), Target T, Subset of variables **CPC**
$assocF = \max_{X \in V} MinAssoc(X; T|\mathbf{CPC})$
$F = \arg\max_{X \in V} MinAssoc(X; T|\mathbf{CPC})$
Output: $\langle F, Fassoc \rangle$

Algorithm 2 $\overline{MMPC}$

Input: Data on variables V ($|V| = p$), Target T
CPC $= \emptyset$
while CPC is still changing **do**
    $\langle F, assocF \rangle = MaxMinHeuristic(T; \mathbf{CPC})$
    if $assocF \neq 0$ **then**
        CPC $\leftarrow$ **CPC** $\cup F$
    end if
end while
for $X \in$ **CPC do**
    if $\exists S \subseteq$ **CPC**, s.t. $Ind(X; T|\mathbf{S})$ **then**
        CPC $\leftarrow$ **CPC** $\setminus \{\mathbf{X}\}$
    end if
end for
Output: Candidate Parent-Child Set **CPC**

This includes the backward step, which we are skipping since having some false positives in the Markov Blanket should not be considered an important problem at this stage of the algorithm. Thus, we will stick with $\overline{MMPC}$.

Algorithm 3 $MMPC$

Input: Data on variables V ($|V| = p$), Target T
CPC $= \overline{\mathbf{MMPC}}$
for $X \in$ **CPC do**
    if $T \notin \overline{MMPC}(X)$ **then**
        CPC $\leftarrow$ **CPC** $\setminus \{X\}$
    end if
end for
Output: CPC

5.1.2 SES

Algorithm 4 SES

Input: Data on variables V ($|V| = p$), Target T, Significance Threshold a, Max Conditioning Set k
$R \leftarrow V$, $S \leftarrow \emptyset$, $Q_i \leftarrow i$, for $i = 1, \ldots, p$
while $R \neq \emptyset$ **do**
  for $X \in R \cup S$ **do**
    if $\exists Z \subseteq S \setminus \{X\}$, $|Z| \leq k$, s.t. $\rho_{XT|Z} > a$ **then**
      $R \leftarrow R \setminus \{X\}$, $S \leftarrow S \setminus \{X\}$
      if $\exists Y \in Z$, s.t. $Z' \leftarrow (Z \cup \{X\}) \setminus \{Y\}$ and $\rho_{YT|Z'} > a$ **then**
        $Q_Y \leftarrow Q_Y \cup Q_X$
      end if
    end if
  end for
  $M = \underset{X \in R}{\operatorname{argmax}} \min_{Z \subseteq S, |Z| \leq k} - \rho_{XT|Z}$
  $R \leftarrow R \setminus \{M\}$, $S \leftarrow S \cup \{M\}$
end while
$E \leftarrow \emptyset$
for $i \in S$ **do**
  $E \leftarrow E \cup \{Q_i\}$
end for
Output: Set of signature sets E

5.1.3 PC-Simple

A simplified version of the PC-algorithm where we only test for conditional independence with the target variable and other nodes. More work must be done here for the benefits of this algorithm.

5.2 Markov Blanket Estimation Procedure

1. Apply `getMB` to estimate neighbors of each target node. Nodes identified in this step are defined as first-order neighbors. This step is implemented in `getAllMBs`.

2. Apply `getMB` again to estimate the neighborhoods of the first-order neighbors. Nodes in these sets are called second-order neighbors. This step is implemented in `getFirstOrderNeighborMBs`, which is called by `constructFinalMBList`

3. If necessary, capture any spouses of the target node (a) and their neighborhoods (b). This step is implemented in `captureSpouses` (a) and `constructFinalMBList` (b).

5.3 `getAllMBs`

Main Tasks

- Provide a wrapper function for obtaining the Markov Blankets of the target nodes and their first-order neighbors

In this function, we intend to recover all of the Markov Blankets for each of the targets, and the information found here will be used to create our Markov Blanket List object for the Local FCI algorithm. First, we apply the function `getMB` to all of the target nodes in order to obtain the Markov Blankets for each target. Upon concluding this step, we call `constructFinalMBList` in order to obtain each target node's second-order neighbors by applying `getMB` to each of the first-order neighbors identified in the previous step. The list we obtain here will be used throughout the algorithm whenever a node's Markov Blanket is desired.

5.3.1 `getMB`

Main Tasks

- The workhorse function of `getAllMBs` to estimate Markov Blankets using algorithms from the `MXM` library for a single target node
-

In this function, we use one of the algorithms provided by `MXM` library to estimate the Markov Blanket of a single target node. As of right now, we have three algorithms: MMPC, SES, and PC-simple.

We return the selected variables and the total runtime for obtaining the values, along with the number of conditional independence tests or association measures taken. For MMPC, based on the algorithm, we only take the number of variables accepted into the candidate PC-set (i.e. parent-child set) plus one as the number of tests, even though there are multiple associations calculated at each step. This calculation requires more work to ensure accuracy and a fair comparison between the algorithms.

5.3.2 `constructFinalMBList`

Main Tasks

- Identify neighborhoods of first-order neighbors
- Capture spouses for PC-set algorithms such as MMPC and SES, and find spouse neighborhoods if necessary
- Combine all neighborhoods lists into one and return the total number of tests and the total time

In this function, we take the nodes identified as first-order neighbors of the target nodes and apply the `getMB` function again to obtain the second-order nodes. Using both of these lists, we then call the `captureSpouses` function if we are using either the MMPC or SES algorithms (since these only capture parent-child sets) to identify which of the second-order neighbors may be spouses and should be included in a target node Markov Blanket.

5.3.2.1 `captureSpouses`

Main Tasks

- Serves as a helper function for `constructFinalMBList`
- Estimate the neighborhoods of first-order neighbors
- Attempts to identify spouses, and their neighborhoods if necessary, for algorithms like MMPC and SES

In this function, we iterate through each of the second-order neighbors for the target nodes and conduct an independence test of the target and the second-order neighbor conditioned on the parent-child set. If we decide in favor of dependence (i.e. reject H_0), then we add the neighbor to the target node's Markov Blanket as a spouse.

5.4 `getEstInitialDAG`

Main Tasks

- Compress neighborhood estimation information from previous steps to prepare for Rcpp code

From the previous steps, we take the list of Markov Blankets around the target nodes and combine the information into an adjacency matrix. In our matrix `adj`, if a node `n` is included in the Markov Blanket of node `m`, then `adj[m,n] = 1` and `adj[n,m] = 1`. This type of representation is preferable for Rcpp code since working with lists is cumbersome and slow in Rcpp.

6 Data Structures

6.1 Graph

Key attributes:

- Provides easy access to information about the adjacency matrix, including adjacent or non-adjacent nodes

The purpose of this class is to provide a base class for storing the graph objects we will work with in our algorithms, whether it be the estimated graph or an inputted reference graph. For the sake of concision and the repeated requests of certain tasks, it made sense to make an underlying class to handle internally everything related to the graphs we are considering.

For the graph class, the most basic variables required are the size of the network, an adjacency matrix, and the names of the nodes. Though all variables could have been given "protected" status, the most essential variables to reveal within the scope of a derived class would be the adjacency matrix itself and the verbose variable. Derived classes should have access to the adjacency matrix mainly due to convenience and simplicity of notation since they too make many repeated requests to either access or set adjacency matrix information.

We also include "sharedFunctions" because it contains a function that helps us to print the adjacency matrix of our graph. The file "sharedFunctions.h" contains a dependency on RcppArmadillo, which will carry throughout the package. For speed improvements, it may be beneficial to make all of the underlying data structures increasingly reliant on this library, but that would be a feature of future research and work.

Most of the member functions simply return basic information about the graph, but the critical function implementations to take note of are the ones which refer to neighbors or adjacent nodes. Note that the `areNeighbors` function returns true if either the (i, j) entry or the (j, i) are not equal to 0. Below, we can also find that we implement `getAdjacent` and `getNonAdjacent` to return nodes that have either one entry not equal to 0 or have both entries equal to 0, respectively. These function definitions are important because they are distinguished from similar functions in the DAG class. While spouses should be considered part of a neighborhood in a graph as well as a DAG, I made the decision to refrain from including them due to an implementation choice in storing Markov Blanket information, which will be seen in a followign section. To maintain simplicity, if one desires to return a neighborhood which includes spouses, it is best to use the DAG class for our particular design.

Please note that, according to our implementation, the constructor immediately creates an adjacency matrix for a complete graph by default, or accepts an adjacency matrix as one of its arguments.
Implementation:

6.2 DAG

Key attributes:

- Allows for simple access to a node's neighborhood (including spouses) and whether or not two nodes belong to the same neighborhood
- Can determine whether or not a node is an ancestor of another node

The DAG class is derived from the Graph class.

There are some specific DAG functions that we need to consider, including the distinction in identifying neighborhoods in a graph versus identifying one in a DAG (i.e. including spouses in the final neighborhood). We also need to identify ancestral relations for the purposes of our metrics in the local algorithm.

In our constructor, nothing substantial is added beyond what we have in the `Graph` class. One primary difference is that if we aren't provided with an adjacency matrix, we begin with an empty graph rather than a complete graph, since we are now dealing with DAGs and a complete graph would be inappropriate. An additional note must be made about the argument `estDAG`, which is an argument specifying whether or not we are dealing with an "estimate of the true DAG" or if we are dealing with the true DAG itself. The distinction is relevant for whether or not we consider spouses to be part of the neighborhood in functions like `inNeighborhood` and `getNeighbors`. An "estimate of the true DAG" refers to a DAG object with an adjacency matrix provided by Markov Blankets which were estimated earlier. In that estimation procedure, we do not identify spouses as such, but merely as neighbors. Therefore, they are not distinguished that way and thus any spouses found in the ensuing matrix are coincidental and should not provide the basis for identifying further neighbors. More details regarding this decision may be found in the section on Markov Blanket estimation. By default, this member variable is set to `false`.

We also have a function to test the acyclicity of the DAG using Kahn's algorithm for a topological sort.

Another important function to consider is the `isAncestor` function. We recursively identify parents of the potential descendant node and all of its parents, checking for membership of the potential ancestor node, until we have no longer any more to consider.

We include the `algorithm` library in order to use the sorting algorithm for our neighborhood identifying functions.
Implementation:

6.3 MBList

Key attributes:

- Facilitates easy access to a node's Markov Blanket (also can be used for a vector of nodes)

This class was created to facilitate access to the Markov Blankets of nodes in which we are interested. There is both a sample version of this and a population version. For the sample version, the matrix is inputted as a result from our estimation procedure, and the Markov Blanket for node i would be identified by all entries equal to 1 in row i of the given matrix. For the population version, we identify Markov Blankets using the usual rules governing identifying parents, children, and spouses, and we use the implementation from the DAG class to identify these nodes.
Implementation of this class:

6.4 SepSetList

Key attributes:

- Facilitates storing and accessing separation sets between variables (`NA` [not separated], `-1` [empty set], otherwise a numeric vector specifying which nodes separate them

Implementation of this class:

7 Constrained Algorithm Class

Key attributes:

- Serves as the basis for the local PC and local FCI classes
- Abstract class that takes care of a lot of the overlapped functions, but still leaves important implementation to derived classes

7.1 Class Constructor

7.2 Check if Variables Are Separated

7.3 Identify V-Structures

8 Local FCI Class

8.1 Class Constructor

8.2 Get Total Skeleton

This is the first skeleton algorithm that we only apply to the targets and their first-order neighbors, obtained from the estimated Markov Blankets or from the inputted true DAG.

8.3 Get Target Skeletons

8.4 Obtain V-Structures

8.5 FCI Rules

Define functions for the rules used in the FCI algorithm Definitions:

0 : No edge

1 : $-o$

2 : $\rightarrow$ (arrowhead)

3 : $-$ (tail)

Note that a $*$ represents a wild card that can represent either an open circle or an arrow When we are considering the PC algorithm, for nodes i and j, if we have $i \rightarrow j$, we regard $G(i,j) = 2$ and $G(j,i) = 3$. Therefore, in keeping with this convention, for FCI we will consider the arrowhead information stored in G to be related to the arrowhead incident on the second node for the edge in consideration. For example, if $io- > j$ is the edge in consideration, $G(i,j) = 2$, since the arrow is incident on j, the second node Similarly, $G(j,i) = 1$, since the circle is incident on node i, which is the second node for the edge between node j and node i.

8.6 Convert Ancestral Graph

8.7 Convert Final Graph

9 Running the Algorithm

10 Conditional Independence Tests

Implementation:

11 Metrics