Acc-pyCausalFS (CTPC) User's Manual in Python

Public domain version 1.0 for Python
Release date: 2021/3/22

Wentao Hu [1, 2], Kui Yu [1, 2], Shuai Yang [1, 2], Xianjie Guo [1, 2], Hao Wang [1,2]

[1] Key Laboratory of Knowledge Engineering with Big Data (Hefei University of Technology), Ministry of Education

[2] School of Computer Science and Information Engineering, Hefei University of Technology, Hefei 230009, China

wentaohu@mail.hfut.edu.cn, yukui@hfut.edu.cn, yangs@mail.hfut.edu.cn, xianjieguo@mail.hfut.edu.cn, sjxwangh@hfut.edu.cn.

We name the project whose algorithms is accelerated by the CItest cache table based PC (CTPC) learning frameworks as Acc-pyCausalFS. Acc-pyCausalFS is a software toolbox for accelerating existing Bayesian Network structure learning algorithms without sacrificing accuracy. It provides the open-source library for in Python.

Copyright © 2021 Wentao Hu, Kui YU, Shuai Yang, Xianjie Guo, Hao Wang.

CONTENTS

1. overview of Acc-pyCausalFS library

1.1 Introduction

The type of constraint-based methods is one of the most important approaches to learn Bayesian network (BN) structures from observational data with conditional independence (CI) tests. In this paper, we find that existing constraint-based methods often perform many redundant CI tests, which significantly reduces the learning efficiency of those algorithms. To tackle this issue, we propose a novel framework to accelerate BN structure learning by reducing redundant CI tests without sacrificing accuracy. Specifically, we first design a CItest cache table to store CI tests. If a CI test has been computed before, the result of the CI test is obtained from the table instead of computing the CI test again. If not, the CI test is computed and stored in the table. Then based on the table, we propose two CItest cache table-based PC (CTPC) learning frameworks for reducing redundant CI tests for BN structure learning. Finally, we instantiate the proposed frameworks with existing well-established local and global BN structure learning algorithms.

The library provides the open-source library for using in Python that implement the state-of-art Bayesian network structure learning algorithms.

1.2 Architecture of Acc-pyCausalFS

The Acc-pyCausalFS architecture is based on two parts, that is, CTPC part, and Application Part. The hierarchy of the Acc-pyCausalFS folder is as shown in Figure 1. The CTPC part consist two subfolders: *"CCT"* and *"PC"* [instantiated PC methods accelerated by CCT]. The Application part consist three folders: *"MB"* [Markov Blanket algorithms], *"LSL"* [local BN structure learning algorithms], and *"GSL"* [global BN structure learning algorithms].

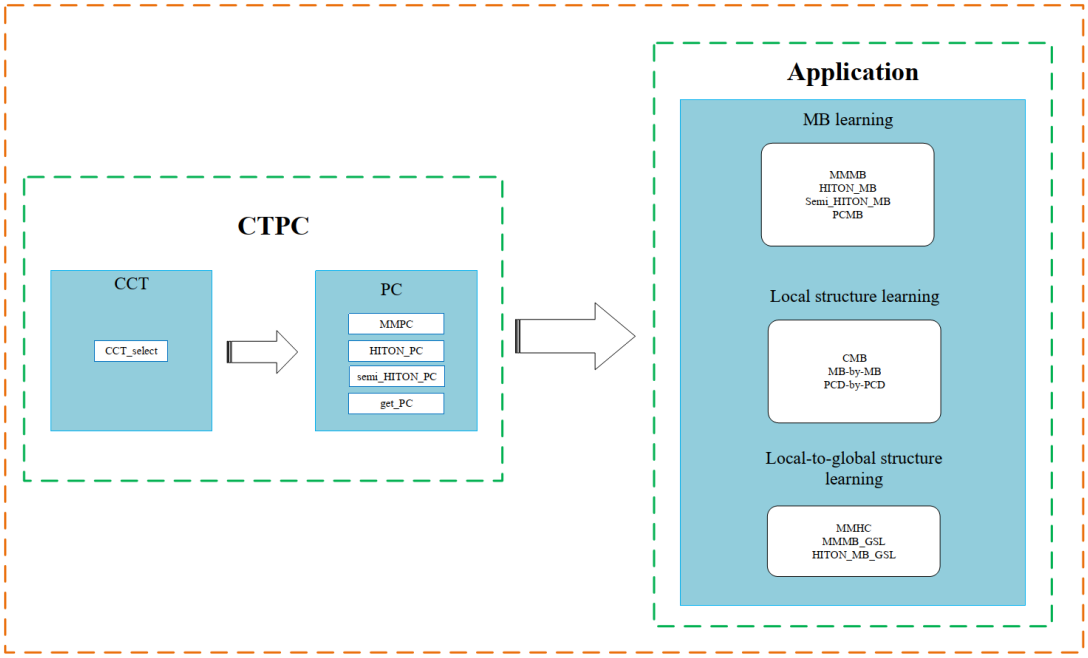

Figure 1 The architecture of Acc-pyCausalFS

The "CCT_select" algorithm is kept in the *"/CTPC/CCT"* folder, which aims to retrieve and store CI tests in the cache table.

In the PC Based on the two proposed CTPC learning frameworks, we have accelerated four PC learning methods, i.e. MMPC、HITON_PC、semi_HITON_PC and getPC, which are stored in the *"/CTPC/PC"* folder.

In the Application part, we have accelerated twelve representative algorithms by the CTPC frameworks. *"/MB"* folder consists of four representative MB learning algorithms, MMMB, HITON-MB, semi-HITON-MB and PCMB; three state-of-the-art local structure learning algorithms are kept in *"/LSL"* folder, i.e. PCD-by-PCD, MB-by-MB, and CMB. *"/GSL"* folder includes three local-to-global structure learning algorithms, MMHC MMMB-CSL and HITON-MB-CSL.

2. CTPC Part

2.1 CCT_select algorithm

Description

Implement the CCT_select algorithm to retrieve and store CI test.

Usage

[pvalue, dep, dict_cache] = CCT_select (data, var1, var2, con_set, is_discrete, dict_cache);

Arguments

Inputs	Data	For an input dataset, columns denote variables and rows represent data observations. The dataset can be continuous or discrete.
	Var1	Index of variable var1 in the data matrix
	Var2	Index of variabel var2 in the data matrix
	Cond_set	Conditioning set. It includes the indices of features in the data matrix.
	Is_discrete	whether an input dataset is discrete or continuous.
	Dict_cache	Store the CItest for reducing redundant CI tests.
Outputs	Pvalue	The p-value of the test
	Dep	The dependency of the test
	Dict_cache	Store the CItest for reducing redundant CI tests.

2.2 Accelerated PC learning algorithms

Description

Accelerating the PC algorithms by the two CTPC frameworks.

Usage

[PC, sepset, ci_number, dict_cache] = getPC (data, target, alpha, is_discrete, dict_cache);
[PC, sepset, ci_number, dict_cache] = HITON_PC (data, target, alpha, is_discrete, dict_cache);
[PC, sepset, ci_number, dict_cache] = semi_HITON_PC (data, target, alpha, is_discrete, dict_cache);
[PC, sepset, ci_number, dict_cache] = MMPC (data, target, alpha, is_discrete, dict_cache);

Arguments

Inputs	Data	For an input dataset, columns denote variables and rows represent data observations. The dataset can be continuous or discrete.
	Target	The target variable
	Alpha	The significance level for conditional independence tests.
	is_discrete	whether an input dataset is discrete or continuous.
	dict_cache	Store the CItest for reducing redundant CI tests.
Outputs	PC	The PC of the target variable.
	Sepset	The sepset of the variable.
	ci_number	The number of CI test performing in the test.
	dict_cache	Store the CItest for reducing redundant CI tests.

3. Application Part

3.1 Accelerated MB, LSL and GSL algorithms

Description

Implement BN structure learning algorithms accelerated by the two CTPC frameworks.

Usage

Initialize a cache table to store CI tests, and then create the key-value pair to count the CI test. dict_cache[‘cache’][0] is to count the number of redundant CI test, while dict_cache[‘cache’][1] is to count the total number of CI test.

dict_cache = {};
dict_cache.setdefault(“cache”,[0,0]);

MB learning algorithms
[MB, ci_number, dict_cache] = MMMB (data, target, alpha, is_discrete, dict_cache);
[MB, ci_number, dict_cache] = HITON_MB (data, target, alpha, is_discrete, dict_cache);
[MB, ci_number, dict_cache] = semi_HITON_MB (data, target, alpha, is_discrete, dict_cache);
[MB, ci_number, dict_cache] = PCMB (data, target, alpha, is_discrete, dict_cache);

#LSL algorithms
[parents, children, PC, undirected, ci_number, dict_cache] = PCDbyPCD (data,target, alpha, isdiscrete, dict_cache);
[parents, children, PC, undirected, ci_number, dict_cache] = PCDbyPCD (data,target, alpha, isdiscrete, dict_cache);
[parents, children, PC, undirected, ci_number, dict_cache] = PCDbyPCD (data,target, alpha, isdiscrete, dict_cache);

GSL algorithms
[DAG, ci_number, dict_cache] = MMHC (data, alpha, dict_cache);

MMMB_GSL
[DAG, ci_number, dict_cache] = MBGSL (data, alpha, isdiscrete, selected=1);
HTION_MB_GSL
[DAG, ci_number, dict_cache] = MBGSL (data, alpha, isdiscrete, selected=2);