Credici Documentation

Release 1.0

Aug 05, 2021

QUICK START

1 Installation 3

2 Citation 5

Credici is an open-source Java library for causal analysis. Inference is done using well-founded methods for inference on credal networks, which is done in a transparently to the user.

The main features of Credici are:

- Allows to easily define Structural Causal Networks (SCM).
- Causal inference: causal effects and counterfactuals.
- Inference is based in methods for inference in credal networks (exact and approximate).
- SCMs can me transformed in equivalent credal networks can be exported in UAI format.

CHAPTER

ONE

INSTALLATION

Credici can be installed from maven. For further details, check the Installation section.

CHAPTER

TWO

CITATION

For the theoretical results in which this tool is based, refer to the following publication.

```
@InProceedings{zaffalon2020structural,
  author    = {Zaffalon, Marco and Antonucci, Alessandro and Caba\~{n}as, Rafael},
  title     = {Structural Causal Models Are Credal Networks},
  booktitle = {Proceedings of the tenth International Conference on Probabilistic␣
↪Graphical Models},
  year      = {2020},
  series    = {Proceedings of Machine Learning Research},
  address   = {Aalborg, Denmark},
  month     = {23--25 Sep},
  publisher = {PMLR},
}
```

2.1 Getting Started

2.1.1 30 seconds to Credici

As a short introduction to Credici, let us consider the following code snippet, in which an structural causal model is built from a discrete Bayesian network. A counterfactual query is performed using an approximate linear programming method.

```
package docs;

import ch.idsia.credici.inference.CredalCausalApproxLP;
import ch.idsia.credici.model.StructuralCausalModel;
import ch.idsia.credici.model.builder.CausalBuilder;
import ch.idsia.crema.IO;
import ch.idsia.crema.factor.credal.linear.IntervalFactor;
import ch.idsia.crema.model.graphical.specialized.BayesianNetwork;

import java.io.IOException;

public class StartingWithCredici {
    public static void main(String[] args) throws IOException, InterruptedException {

        // Load the empirical model
        BayesianNetwork bnet = (BayesianNetwork) IO.read("models/simple-chain.uai");

        // Build the causal model
        StructuralCausalModel causalModel = CausalBuilder.of(bnet).build();
```

(continues on next page)

(continued from previous page)

```
        // Set up the inference engine
        CredalCausalApproxLP inf = new CredalCausalApproxLP(causalModel, bnet.
↪getFactors());

        // Run the query
        IntervalFactor res = (IntervalFactor) inf.counterfactualQuery()
                .setTarget(2)
                .setIntervention(0,0)
                .setEvidence(2, 1)
                .run();

    }
}
```

2.2 Requirements

2.2.1 System

Credici requires Java 11 or higher and maven (https://maven.apache.org). Tests have been done under Linux Ubuntu and macOS with openjdk 11 and 12.

2.2.2 Package Dependencies

Credici contains the dependencies shown below which are transparently managed with maven.

- org.apache.commons:commons-lang3:jar:3.4:compile
- ch.idsia:crema:jar:0.1.4-SNAPSHOT:compile
- org.eclipse.persistence:org.eclipse.persistence.moxy:jar:2.6.2:compile
- com.google.code.findbugs:jsr305:jar:3.0.2:compile
- commons-cli:commons-cli:jar:1.4:compile
- org.checkerframework:checker-qual:jar:2.10.0:compile
- org.junit.jupiter:junit-jupiter-api:jar:5.4.2:test
- org.jgrapht:jgrapht-core:jar:1.1.0:compile
- com.google.guava:listenablefuture:jar:9999.0-empty-to-avoid-conflict-with-guava:compile
- javax.xml.bind:jaxb-api:jar:2.3.1:compile
- org.apache.commons:commons-csv:jar:1.3:compile
- com.google.errorprone:error_prone_annotations:jar:2.3.4:compile
- org.opentest4j:opentest4j:jar:1.1.1:test
- com.google.guava:guava:jar:28.2-jre:compile
- org.glassfish:javax.json:jar:1.0.4:compile
- org.eclipse.persistence:org.eclipse.persistence.asm:jar:2.6.2:compile
- com.google.j2objc:j2objc-annotations:jar:1.3:compile

- net.sourceforge.csparsej:csparsej:jar:1.1.1:compile
- org.apache.commons:commons-math3:jar:3.6.1:compile
- colt:colt:jar:1.2.0:compile
- com.google.guava:failureaccess:jar:1.0.1:compile
- org.hamcrest:hamcrest-core:jar:1.3:compile
- log4j:log4j:jar:1.2.14:compile
- com.joptimizer:joptimizer:jar:3.5.1:compile
- net.sf.lpsolve:lp_solve:jar:5.5.2:compile
- junit:junit:jar:4.12:compile
- com.github.quickhull3d:quickhull3d:jar:1.0.0:compile
- javax.activation:javax.activation-api:jar:1.2.0:compile
- commons-logging:commons-logging:jar:1.2:compile
- concurrent:concurrent:jar:1.3.4:compile
- org.junit.platform:junit-platform-commons:jar:1.4.2:test
- org.junit.jupiter:junit-jupiter-params:jar:5.4.2:test
- javax.validation:validation-api:jar:1.1.0.Final:compile
- net.sf.trove4j:trove4j:jar:3.0.3:compile
- ch.javasoft.polco:polco:jar:4.7.1:compile
- org.eclipse.persistence:org.eclipse.persistence.core:jar:2.6.2:compile
- org.slf4j:slf4j-api:jar:1.7.7:compile
- org.springframework:spring-core:jar:2.5.6:compile
- org.apiguardian:apiguardian-api:jar:1.0.0:test

2.3 Installation

Credici can be easily included at any maven project. For this, add the following code in the pom.xml:

```
<repositories>
    <repository>
        <id>cremaRepo</id>
        <url>https://raw.github.com/idsia/crema/mvn-repo/</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>ch.idsia</groupId>
        <artifactId>credici</artifactId>
        <version>0.1.3</version>
        <scope>compile</scope>
    </dependency>
</dependencies>
```

2.4 Causal Model Definition

Here we will consider the different ways for defining a structural causal model (SCM) in Credici. This can be done by explicitly specifiying all the nodes, arcs and factors in the model, or with the help of the class `CausalBuilder`.

2.4.1 Explicit Definition

The code snippet shown below shows how to explicitly define a SCM. For this, an object of class `StructuralCausalModel` is created. Then endogenous and exogenous variable are added to the model by indicating the cardinality. In case of the exogenous ones, the second input parameter should be set to `true`, which indicates the type of variable. Then the parents are set and finaly the factors are specified, which are basically objects of class `BayesianFactor`.

```
StructuralCausalModel model = new StructuralCausalModel();

// define the variables (endogenous and exogenous)
int x1 = model.addVariable(2);
int x2 = model.addVariable(2);
int x3 = model.addVariable(2);
int x4 = model.addVariable(2);

int u1 = model.addVariable(2, true);
int u2 = model.addVariable(4, true);
int u3 = model.addVariable(4, true);
int u4 = model.addVariable(3, true);

// Define the structure
model.addParents(x1, u1);
model.addParents(x2, u2, x1);
model.addParents(x3, u3, x1);
model.addParents(x4, u4, x2, x3);


// define the CPTs of the exogenous variables
BayesianFactor pu1 = new BayesianFactor(model.getDomain(u1), new double[] { .4, .6 });
BayesianFactor pu2 = new BayesianFactor(model.getDomain(u2), new double[] { .07, .9, .
↪03, .0 });
BayesianFactor pu3 = new BayesianFactor(model.getDomain(u3), new double[] { .05, .0, .
↪85, .10 });
BayesianFactor pu4 = new BayesianFactor(model.getDomain(u4), new double[] { .05, .9, .
↪05 });

model.setFactor(u1,pu1);
model.setFactor(u2,pu2);
model.setFactor(u3,pu3);
model.setFactor(u4,pu4);

// Define the CPTs of endogenous variables as deterministic functions
BayesianFactor f1 = EquationBuilder.of(model).fromVector(x1, 0,1);
BayesianFactor f2 = EquationBuilder.of(model).fromVector(x2,0,0,1,1,  0,1,0,1);
BayesianFactor f3 = EquationBuilder.of(model).fromVector(x3,0,0,1,1,  0,1,0,1);
BayesianFactor f4 = EquationBuilder.of(model).fromVector(x4,0,1,1,  0,0,0,  0,0,0, 0,
↪1,1);

model.setFactor(x1,f1);
```

(continues on next page)

(continued from previous page)

```
model.setFactor(x2,f2);
model.setFactor(x3,f3);
model.setFactor(x4,f4);

model.printSummary();
```

2.4.2 Causal Builder

Additionaly, Credici provides class `CausalBuilder` at `ch.idsia.credici.model.builder` for simplifying the code under some settings. We will assume that we have `BayesianNetwork` specifying the empirical information: empirical DAG and eventually empirical probabilities. For example:

```
BayesianNetwork bnet = new BayesianNetwork();
int y = bnet.addVariable(2);
int x = bnet.addVariable(2);
```

The following code shows 4 equivalent ways of building a SCM from such BN under the markovian setting.

```
// Markovian equationless from BN
StructuralCausalModel m1 = StructuralCausalModel.of(bnet);

// Markovian equationless from DAG and sizes
StructuralCausalModel m2 = StructuralCausalModel.of(bnet.getNetwork(), bnet.
↪getSizes(bnet.getVariables()));

// Markovian equationless from BN
StructuralCausalModel m3 = CausalBuilder.of(bnet).build();

// Markovian equationless from DAG and sizes
StructuralCausalModel m4 = CausalBuilder.of(bnet.getNetwork(), bnet.getSizes(bnet.
↪getVariables())).build();
```

In the previous cases, factors associated to exogenous variables are empty. Instead, we could build it with some random factors:

```
// Markovian equationless with random P(U)
StructuralCausalModel m5 =
        CausalBuilder.of(bnet)
                .setFillRandomExogenousFactors(2)
                .build();

// Markovian with random P(U) and equations
StructuralCausalModel m6 =
        CausalBuilder.of(bnet)
                .setFillRandomExogenousFactors(2)
                .setFillRandomEquations(true)
                .build();
```

Instead of considering the default structural equation, these could be specifyed as follows.

```
// Markovian case specifying equations and with random exogenous factors

BayesianFactor eqy = EquationBuilder.fromVector(
        Strides.as(y,2), Strides.as(u1,2),
```

(continues on next page)

(continued from previous page)

```
        0,1
);

BayesianFactor eqx = EquationBuilder.fromVector(
        Strides.as(x,2), Strides.as(u2,4),
        0,1,1,0
);

BayesianFactor[] eqs = {eqy, eqx};

StructuralCausalModel m9 = CausalBuilder.of(bnet)
        .setEquations(eqs)
        .setFillRandomExogenousFactors(3)
        .build();
```

The quasi-markovian case could be also considered by specifying the causal DAG, which should be consistent with the empirical one.

```
// Quasi Markovian specifying causal DAG with random factors
SparseDirectedAcyclicGraph dag2 = bnet.getNetwork().copy();
int u = dag2.addVariable();
dag2.addLink(u, y);
dag2.addLink(u,x);

StructuralCausalModel m8 =
        CausalBuilder.of(bnet)
                .setCausalDAG(dag2)
                .setExoVarSizes(new int[]{4})
                .setFillRandomEquations(true)
                .build();
```

2.5 Causal Inference

2.5.1 Credal Network Transformation

Any object of class `StructuralCausalModel` can be converted into an equivalent credal network using the methods `toVCredal` and `toHCredal` for a vertex and a constraints specification. The input is a collection of *BayesianFactors*' which are the empirical distributions.

```
// convert the causal models into credal networks
SparseModel vcredal = causalModel.toVCredal(bnet.getFactors());
SparseModel hcredal = causalModel.toHCredal(bnet.getFactors());
```

2.5.2 Inference engine

First the exact and approximate inferences engines should be set up. For this create instances of classes `CredalCausalVE` and `CredalCausalAproxLP` as shown in the following code snippet.

```
// set up the exact inference engine
CredalCausalVE infExact = new CredalCausalVE(causalModel, bnet.getFactors());
// set up the approximate inference engine
CredalCausalApproxLP infApprox = new CredalCausalApproxLP(causalModel, bnet.
↪getFactors());
```

(continues on next page)

(continued from previous page)

Alternatively, engines can be instantiated from a credal network.

```
// set up the exact inference engine
CredalCausalVE infExact = new CredalCausalVE(vcredal);
// set up the approximate inference engine
CredalCausalAproxLP infApprox = new CredalCausalAproxLP(hcredal);
```

2.5.3 Causal Effects

Let us consider the causal effect of on a variable X_3 of a variable $X_1 = 1$, that is, $P(X_3|do(X_1 = 1))$. This can be calculated with the exact inference engine as follows.

```
// set up and run a causal query
VertexFactor resExact = (VertexFactor) infExact
        .causalQuery()
        .setTarget(x[3])
        .setIntervention(x[1],1)
        .run();
```

Alternatively, for an approximate solution:

```
// set up and run a causal query
IntervalFactor resApprox = (IntervalFactor) infApprox
        .causalQuery()
        .setTarget(x[3])
        .setIntervention(x[1],1)
        .run();
```

2.5.4 Conuterfactuals

Credici also allows counterfactual queries (in a twin graph) such as $P(X'_3|do(X'_1 = 1), X'_1 = 0)$. The exact computation of this query can be done as follows.

```
// exact inference
resExact = (VertexFactor) infExact
        .counterfactualQuery()
        .setTarget(x[3])
        .setIntervention(x[1],1)
        .setEvidence(x[1], 0)
        .run();
```

On the other hand, using the approximate engine:

```
// set up and run a counterfactual query
resApprox = (IntervalFactor) infApprox
        .counterfactualQuery()
        .setTarget(x[3])
        .setIntervention(x[1],1)
        .setEvidence(x[1], 0)
        .run();
```

2.6 Contact and Support

Credici has been developed at the Swiss AI Lab IDSIA (Istituto Dalle Molle di Studi sull'Intelligenza Artificiale). The members of the development and research team are:

- Rafael Cabañas (rcabanas@idsia.ch)
- Alessandro Antonucci (alessandro@idsia.ch)
- David Huber (david@idsia.ch)
- Marco Zaffalon (zaffalon@idsia.ch)

If you have any question, please use Github issues.