Learning Causal Networks in Python

James Callan

Contents

1 Introduction

The aim of this report is to detail the implementation of various algorithms to learn causal networks in python.These algorithms have already been implemented in the statistical programming environment R. However, recently python has become more widely used in the field of statistical computation.

Python offers a number of advantages over R. Python is much more widely used in general than R so an implementation in python would be accessible to a wider array of developers.Python is also used in a variety of settings as it is a general purpose programming language. This would allow the software implementation to be integrated into many different systems, written in python, with relative ease.

2 Literature Review

2.1 Causal Networks

Causal Networks are graphical models which represent a set of variables, their conditional dependencies, and their causal relationships [1] as a Directed Acyclic Graph (DAG) or a Maximal Ancestral Graph (MAG). The nodes of the Graph represent the variables. The edges of the graph represent causality, the direction of the edge represents the direction of causality with parent nodes causing child nodes [1].

In a DAG all edges have a single direction, whereas in a MAG edges may be directed or bidirectional [2].

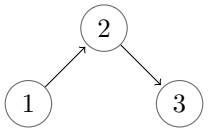


Figure 1 A DAG with 3 nodes and 2 edges

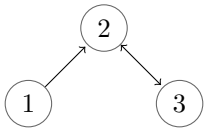


Figure 2 A MAG with 3 nodes and 2 edges

The learning of causal networks allows relationships between variables to be uncovered and presented in a simple and human readable fashion. This can provide useful information for further data analysis as the set of variables that

cause another could be used to predict it's value.

2.2 Probability and Independence

In probability theory we can quantify our confidence in a particular event E occurring. This confidence, denoted as $P(E)$ is a real number between 1 and 0. $P(E) = 1$ representing a certainty and $P(E) = 0$ representing E being impossible.

In the case of random variables an event would be a variable X taking a particular value x_i, where x_i is a member of the set $x = \{x_1, x_2, ..., x_n\}$ of possible values X can take. The probability of this event is denoted as $P(X = x_i)$.

The probability distribution $P(X = x)$ or simply $P(X)$ defines the probabilities for every value which X can take.

Distributions of more than one variable can be described with joint distributions. For Variables X and Y the distribution $P(X, Y)$ describes the probability of both X and Y taking particular values simultaneously.

Joint distributions can also describe how the value of one variable can effect the value of an other. $P(X|Y)$ describes the probabilities of X taking particular values "given" a value that Y has taken. $P(X|Y)$ is defined as $P(X, Y)/P(Y)$.

If two variables are independent there is no correlation between the values they take. That is for two variables X and Y, the distribution $P(X|Y)$ would be the same for all values of Y. Therefore $P(X|Y) = P(X)$. If two variables are independent we can determine that there is no causal relationship between them.

Given the definition $P(X|Y) = P(X, Y)/P(Y)$ and $P(X|Y) = P(X)$ when X and Y are independent, it easy to see that $P(X, Y) = P(X)P(Y)$ if X and Y are independent.

Two variables X and Y can be considered independent conditioned on a third variable Z if there is no correlation between X and Y for given the value of Z.

In this case $P(X|Y, Z) = P(X|Z)$, as the value of Y has no impact on the value of X. The definition of conditional distributions shows $P(X|Y, Z) = \frac{P(X,Y,Z)}{P(Y,Z)}$, and $P(Y, Z) = P(Y|Z)P(Z)$. Using theses definitions and basic algebra $P(X, Y|Z) = P(X|Z)P(Y|Z)$ can be shown when X and Y are conditionally independent on Z.

2.3 D Separation

A path is any sequence of adjacent edges regardless of their directionality. A collider is node in a path which is both entered and left on edges which are directed toward the node. Unblocked refers to a path that does not traverse a collider [3].

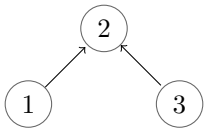


Figure 3 A graph containing a collider

If every path between nodes X and Y traverses a collider, nodes X and Y are unconditionally d-separated or d-separated conditioned on the empty set [3]. Two unconditionally d-separated nodes in a causal network are considered to be independent [4].

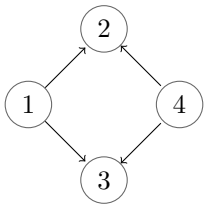


Figure 4

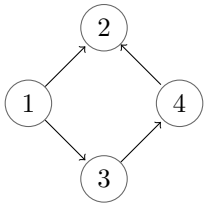


Figure 5

Two nodes X and Y are d-separated conditioned on set Z if:

1. There exists no unblocked path from X to Y that does not traverse any members of Z. [3]
2. There is no blocked path from X to Y in which all colliders are members of Z or have descendants in Z. [3]

Causal networks represent conditional independence with d-separation, variables X and Y are conditionally independent on set Z if nodes X and Y are d-separated conditioned on nodes in set Z [1].

2.4 Faithfulness

By testing for d-separation of variables X and Y given conditioning set Z on a graph we can see if $X \perp\!\!\!\perp Y|Z$ [4]. However, the reverse is only true if the distribution is faithful to the graph. That is, if a distribution is faithful to a graph then all independence relationships are represented on the graph and only these relationships are present [5]. Assuming faithfulness allows the topology of a graph to be learned by testing pairs of variables for independence given various conditioning sets.

2.5 Discriminating paths

For the FCI and RFCI algorithms the notion of discriminating paths is needed. A path π = (A, .., X,Y,Z) is discriminating a discriminating path for Y if:

1. π must include at least 3 edges
2. Y is a non-endpoint on π and is adjacent to Z on π.
3. A is not adjacent to Z and every other node is a collider on π and a parent of Z. [6]

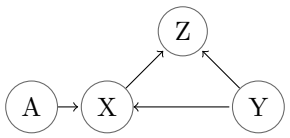


Figure 6
(A,X,Y,Z) is a discriminating path on Y

2.6 Tests for Independence

When trying to find causal relations between variables tests for independence and conditional independence must be performed. These tests allow us to find correlation between variables and also find out if the correlation between two variables is actually caused by another.

One such test is the χ^2 test, this test gives the probability that some data was drawn from a particular distribution.

If the variables X and Y are independent then $p(X, Y) = p(X)p(Y)$. Both of these distributions can be calculated from data. The χ^2 test can then be used to find the probability that they are equal.

To estimate $p(X, Y)$ the counts of all pairs of values taken by the variables can be found. They can then divided by the count of all data points. These counts

can be stored in a table with the values of X as columns, values of Y as rows (or visa versa) and entries as observed counts, for ease of access. This type of table is known as a contingency table.

X	Y
1	0
1	1
0	1
1	1
0	0
1	1
1	0

		X	
		0	1
Y	0	1	2
	1	1	3

To estimate the distribution $p(X)$, the count of points in which X takes a value over the total number of points for all values of X. The counts can be found with the sum of the columns of the contingency table. The same can be done for $p(Y)$ with the rows of the table.

The χ^2 statistic can then be calculated with:

$$\chi^2 = \sum \frac{(p(X)p(Y) - p(X,Y))^2}{p(X)p(Y)}$$

The χ^2 value can then be passed to a cumulative density function and a p-value returned. The function used is dependent upon the degrees of freedom. The degrees of freedom are equal to the $(no.rows - 1) * (no.columns - 1)$. The p-value represents how likely it is that the two variables are independent.

The test is similar for conditional independence. The chi^2 statistic now depends on conditional distributions. If $X \perp\!\!\!\perp Y|Z$ $p(X, Y|Z) = p(X|Z)p(Y|Z)$. Therefore:

$$\chi^2 = \sum \frac{(p(X|Z)p(Y|Z) - p(X,Y|Z))^2}{p(X|Z)p(Y|Z)}$$

These values can all be found by calculating a contingency table for $X, Y and Z$.

This statistic can then be used in the same way as before to calculate a p-value. However, the degrees of freedom are multiplied by the possible values of Z -1.

2.7 Algorithms For Learning Causal Networks

2.7.1 Constraint Based Methods

There have been a number of algorithms developed to learn graphical causal representations from data. Over the course of this project, I plan to implement: The PC algorithm [7], the Fast Causal Inference (FCI) algorithm [6], and the

Really Fast Causal Inference (RFCI) Algorithm [6].

Rather than learning an individual DAG the PC algorithm learns a set of DAGs which could represent the causal relationships found in the data. In cases where the direction of an edge is unclear no direction is specified. The output of the algorithm is a partially directed acyclic graph which represents the set DAGs with all possible orientations of undirected edges [7].

The FCI and RFCI algorithms learn partial ancestral graphs (PAGs) which represent the set of MAGs that fit the data [6]. Unlike in the PC algorithm if the direction of an edge can not be determined it is assumed that there is some set of latent variables influencing the nodes being tested. Each end of an edge can be one of three things in a PAG, an arrow in every MAG ($>$), a tail in every MAG ($-$) or an arrow in at least one MAG and a tail in at least one MAG (o), ends of an edge denoted by $*$ can be any of the specified types. This allows for a richer display of information than a partial DAG [6].

The PAG generated by the FCI algorithm is considered to be less accurate than the PAG generated by the RFCI algorithm. This is because the FCI algorithm performs more tests than the RFCI algorithm. These tests give us more confidence in the presence and the orientation of edges however this comes at the cost of speed.

2.7.2 Score Based Methods

The above algorithms are all constraint based. There is another class of algorithm which are score based. Rather than finding graphs using the rules that define the networks, score based methods generate random graphs and assign them a score based on how well they fit data.

An example of a scored based method is the graphical lasso algorithm.

2.8 Computing a skeleton

All three algorithms begin by calculating the skeleton of the graph. The skeleton consists of the edges that are present in the partial DAG or PAG however all edges are undirected. The separation set of two variables ($sepSet(x, y)$) contains the smallest set of variables that the pair are conditionally independent on. When calculating the skeleton the separation set of each pair of variables is also recorded as it is needed to orient the edges of the graph [6, 7].

The skeleton is calculated using a series of conditional independence tests. These conditional independence tests are dependent on the type of input data, therefore conditional independence tests can be defined independently of the algorithm. The conditional independence tests must be able to determine if two

variables are indpendent based on some set of other variables from data.

To begin the skeleton computation the fully connected undirected graph of all variables is constructed and then conditional independence tests are performed to determine which edges to remove.

Adjacent pairs of variables are tested for independence conditioned upon set of variables adjacent to them. The size of these sets starts at 0 but is incremented after all adjacent pairs have been tested. This repeats until there are not enough adjacent variables to fill the conditioning set.

The edges which have not been removed form the skeleton of the graph which is needed for the PC, FCI and RFCI algorithms along with the recorded separation sets of removed edges [6, 7].

Algorithm to Estimate Skeleton

Input: Data, Independence Test
Output: Estimated Skeleton, Separation Sets of Variables

```
k = 0
N = list of variables in data
G = fully connected undirected graph with nodes = N

while there is some S ∈ adj(X, G) \ Y where |S| = k for some X in N and Y in adj(X,G)
  for all X in N:
    for all Y in adj(X, G):
      for all S ∈ adj(X, G) \ Y where |S| = k:
        if X ⫫ Y | S
          remove edge X,Y from G
          sepSet(X,Y) = S
          break
        end if
      end for
    end for
  end for
  k += 1
end while

return G, sepSet
```

2.9 The PC Algorithm

After generating the skeleton and separation sets of pairs of nodes, the pc algorithm attempts to orient the undirected edges of the graph.

It begins by searching for colliders. For each triple of vertices A, B, and C where A and B are adjacent, B and C are adjacent, but A and C are not adjacent orient the edges A-B-C to A→B←C if B is not in the separation set of the pair (A,C).

Once colliders are specified other edges can be oriented. This is done by repeating two steps until no more edges can be given a direction. The algorithm for edge orientation is described in.

PC Algorithm to Orient Edges

Input: Skeleton of Causal Network, Separation sets of variables
Output: Partial Directed Acyclic Graph

G = Skeleton of network
N = nodes of G

for all i, j, k in N:
  if k **not in** $adj(i, G)$ **and** j **in** $sepSet(i, k)$:
    orient $i - j - k$ as $i \rightarrow j \leftarrow k$
  end if
end for

while no more edges can be oriented:
  for all i, j, k in N:
    if k **not in** $adj(i, G)$:
      orient $i \rightarrow j - k$ as $i \rightarrow j \rightarrow k$
    end if
    if $directedpath(i, j)$ **in** G:
      orient $i - j$ as $i \rightarrow j$
    end if
  end for
end while

return G

2.10 The FCI algorithm

The FCI algorithm starts in the same way as the PC algorithm using exactly the process of conditional independence tests to estimate a skeleton of the final graph. It also orients colliders in the same way, producing a PDAG.

Next, the final skeleton of the graph is obtained from the PDAG G. To do this The possible d-separators of every variable must be found in G. For variable X, the possible d separators set in nG ($PossibleDSep(X, G)$) consists of all variables Y for which there is some path $\pi = path(X, Y)$ in G, such that in every sub path A, B, C, B is a collider or A, B, C form a triangle. All adjacent variables (X, Y) are the tested for conditional independence, conditioned on every subset of each of the possible d separators set. If the test finds conditional independence the edge X, Y is removed and the conditioning set recorded as a separation set.

The PDAG can then be converted to a PAG by setting all edges to $o - o$.

Next the colliders must be reoriented. This is done on a similar way however now edges can be bi-directional.

Once the final skeleton has been estimated, A series of orientation rules can be performed repeatedly until no more orientations can be found. Theses rules where originally described in. However these rules did not fully orient the $-$ tags of the graph [8]. 10 rules were found in [8] which allow full orientation of the graph. The complete set of rules is used in the R implementation and will also be used in the python implementation.

FCI Algorithm to Orient Edges

Input: Skeleton of Causal Network, Separation sets of variables
Output: Partial Ancestral Graph

G = Skeleton of network
N = nodes of G

for all i, j, k in N:
  if k **not in** $adj(i, G)$ **and** j **in** $sepSet(i, k)$:
    orient $i - j - k$ as $i \rightarrow j \leftarrow k$
  end if
end for

for all i in N and j in adj(i,G):
  for all $T \subseteq PossibleDSep(i, G) \setminus \{i, j\}$:
    if $i \perp\!\!\!\perp j | T$:
      remove edge i,j from G
      $sepSet(i, j) = T$
      $sepSet(j, i) = T$
      break
    end if
  end for
  for all $T \subseteq PossibleDSep(j, G) \setminus \{i, j\}$:

```
    if i ⫫ j|T:
      remove edge i,j from G
      sepSet(i, j) = T
      sepSet(j, i) = T
      break
    end if
  end for
end for

set all edges in G to o-o

for all i, j, k in N:
  if k not in adj(i, G) and j in sepSet(i, k):
    orient i ∗ − ∗ j ∗ − ∗ k as i∗ → j ← ∗k
  end if
end for

while no more edges can be oriented:
  Apply orientation rules from [8]:
end while

return G
```

2.11 RFCI Algorithm

Like the other two algorithms the RFCI begins by estimating the skeleton of the final graph.

Next the colliders in the graph are found, this is done differently than in the two previous algorithms. We created and empty list L and for each triple (A,B,C) in the list of unshielded triples we check if both:

i $A \perp\!\!\!\perp B \mid ((sepset(A, B) \setminus \{C\}))$

ii $B \perp\!\!\!\perp C \mid ((sepset(A, B) \setminus \{C\}))$

where sepset(X,Y) is the separating set of Nodes X and Y. If both of these tests hold, add to list L. If only the first test holds, find the minimal separating set of A and B and record it as the separating set of A and B. Next, remove the edge A-B, add any newly create unshielded triples to M, and remove any destroyed triples from L and M. If only the second test passes do the same as above but for nodes B and C.

Finally the edges must be oriented. To begin edge orientation apply rules $R1-R3$ found in [8]. Next triangles (X,Y,Z) must be found with edges $Yo-{*}Z$,

$X <-* Y$, and $X --> Z$. Once a triangle is found the shortest discriminating path from X to Z must be found.

If a path π exists take each pair of adjacent variables A and B on π and creator a counter $l = 0$. Next find a subset Y $\subseteq (sepset(X, Y) \setminus \{A, B\})$ where $|Y| = l$. Then if $A \perp\!\!\!\perp B | Y \cup S$ let sepset(A,B) = Y, delete the edge $A *-* B$ from the graph and update newly formed unshielded triples orientation using the collider orientation described earlier. If the conditional independence test fails increment l and try again, testing with all subsets Y of size l, repeat this process until no set Y exists where —Y— = l or the edge $A *-* B$ is removed.

If no edges are removed from π by the above process then either: orient $Y o-* Z$ as $Y --> Z$ if Y is in the sperating set of X and Z or orient $X <-* Y o-* Z$ as $X <-> Y <-> Z$ if not.

Finally orient as many edges using rules $R5 - R10$ from [8].

This whole process must be repeated until it no longer alters any orientations. [6]

3 Design

3.1 Methodology

When Designing software there are a number of approaches that can be taken. Most of these approaches break development down into a number of different phases and describe the order in which they should be completed.

Most methodologies were designed for teams of engineers, as this project was completed by one individual many components of the methodologies are redundant. However there are some principles and techniques which can be utilised by an individual developer and aid in organisation of a project.

3.1.1 Waterfall

The waterfall methodology is a linear approach to development. Each stage of development is completed all at once for the whole piece of software. This means that once a stage is completed there is no need to return to it and the development process flows in one direction like a a waterfall.

The waterfall approach is good because the whole project is rigorously planned and documented before coding begins. This allows major difficulties in projects to be dealt with before significant time has been invested. It also allows projects to be easily passed between developers as a new developer can look at the plan of the project. In less linear approaches there may be plans for future parts of software which are never documented.

The downfall of the waterfall approach comes when a project is not fixed or there is a a change in requirements, there is no stage in which plans can be modified as that would involve going backwards in development.

3.1.2 Agile

The agile approach is a methodology which focuses on iterative development. Deliverables are identified and designed individually, there is also a much greater focus on executable code than documentation than in waterfall

Many parts of the agile methodology are not needed in this project such the parts that deal with the customer and cooperation between developers as this is a solo project without a customer.

The most important part of agile is its ability to deal with changing scope through process. Also since testing is done at each stage it will allow the project to be easily shortened or extended based on time constraints, as after every sprint a working piece of software has been developed.

3.1.3 Spiral

The spiral model for software development is a risk driven process. The spiral method can contain aspects from the other processes.

The actions taken and the amount of work done on areas of the project are determined by what would minimize risk.

Risk can be a number of things. In an industrial setting risk could be how how spending more time on a product and delaying it's release would affect it's sales figures. In a team project, risks could consist of members of the team being unable to work on software unexpectedly because of illness. In essence, risk is anything that would be detrimental to the stakeholders in the project.

Spiral development also requires risks to be assessed at every stage of development. This would increase the work load specifically in agile as risks would need to be identified at every iteration. This would lead to a lot of repeated work that could be better spent on development or testing.

3.1.4 Chosen Approach

The chosen approach for this project was agile. The reduced amount of documentation will speed up development. Agile development also allows the scope of the project to change over time. This allowed the scope of the project to start off small and increase if time allowed it.

For example, once the PC algorithm had been implemented, if time allowed, the project could easily be extended to include the FCI algorithm.

The spiral approach will not be used for this project as it will lead to an unnecessary increase in work for a relatively low risk project. The project also has a fairly logical order of implementation so using risk minimization to determine what to implement will not be necessary.

Within the Agile methodology there are a number of popular frameworks. However, these teams are designed for teams of software engineers with clients. Since this is a solo project without a client many parts of these frameworks will not be applicable. Therefore, the applicable parts of various frameworks were chosen and compiled into a framework that will work for this project.

3.2 Requirements

Requirements were elicited at the beginning of a sprint. A deliverable was chosen and the requirements were found for that deliverable. Requirements were found by considering how the software would be used.

3.2.1 Epics

Epics describe a very high level interaction between a user and the software. They describe the general functionality of a deliverable without discussing it's implementation. Epics were the first stage of requirement elicitation as they they capture the functionality that the requirements must capture.

3.2.2 User Stories

Each epic is then broken down into user stories. The stories describe what a user must be able to do with the software during the epic. They are fairly low level and form a set of required functionality of the software.

Users stories can be used as requirements as software that would allow user stories to occur would be functional.

3.3 Sprints

Sprints are sections of development defined by deliverables. They outline the tasks required to be completed to develop a deliverable and the time frame needed.

When a sprint is completed, the scope of the project and the requirements can be updated and modified if needed.

for this project there were four main deliverables implemented and four sprints to implement them.

The first sprint consisted of implementing the χ^2 conditional independence test. This test was needed before anything else could be properly implemented.

Next, the skeleton learning software was written. The skeleton is needed for all algorithms in this project so is the next logical step from the independence test.

Then the pc algorithm was fully implemented, this algorithm was the simplest of the 3 so made sense to start with. Starting with the easiest algorithm gave me experience using networkx and the other libraries being used that would be very useful in the final sprint.

Finally, the FCI algorithm was implemented. This was more complex than the pc algorithm, however the experience with the library in previous sprints greatly helped with the more complex orientation.

3.3.1 Tasks

From the stories can be formed. Theses tasks describe the software that must be implemented to fulfil the requirements.

Each sprint had a number of associated tasks. The tasks were small and each task could be verified through testing.

Each task has associated stories so that all software can be traced back to an epic. Tracing back to an epic shows that each task will in some way contribute to functional software.

3.4 Architecture

Only the architecture for a particular deliverable was designed in each sprint. This meant that the architecture grew over time. Considerations for general extensibility of the software needed to be made in the design phase to ease design and implementation during later sprints.

A class diagram was produced, this specified the relationships between classes and the functionality of each class. This diagram evolved throughout the project as various sprints were completed.

The class diagram for this software was fairly straightforward. This is because the software did not implement a complex system. Only a few classes were needed as most of the complexity is contained in the methods of the classes.

3.4.1 Unit Testing

Test driven development (TDD) was one of the main features of the development framework. In this development, tests were created before the actual software was written. This ensured that all software that was written would be functional. It also ensures that any refactoring performed would not compromise the functionality of software.

All software was tested using the python unittest framework. In this framework, list of tests can be compiled into test cases and cases compiled into a test suite. Test cases test an individual feature of software, so can be run individually when a feature is being developed or maintained. Test Suites can then test the software as a whole.

3.5 Documentation

All software written was thoroughly documented. Every class, method and function was given an attached doc string detailing it's functionality and use.

NumPy style documentation was used. An example is shown in —. This style was chosen because it gives a useful summary of the functionality of a piece of code. It also allows other developers to easily use the code by showing exactly what arguments are taken and exactly what will be returned by a function.

""" A function to build the set of conditioning sets to be for variables x and y on graph g of a certain size when generating a skeleton

Parameters
– – – – – – – – – –
X : str,
One variable being tested for independence
Y : str
The other variable being tested for independence
graph : float, optional
The minimum p-value returned by the indepence test
for the data to be considered independent
size: int
the size of each conditioning set to be returned
Returns
– – – – – – –
list of lists of strings
a list of conditioning sets to be tested
"""

Inline comments were also used to clarify areas of code within methods or func-

tions. These comments can allow a developer to quickly and easily understand the code for the purposes of extension or maintenance.

3.6 Tools

A number of existing tools were used throughout the project to aid in development.

3.6.1 Version Control

An important part of software development is Version Control (VC). VC allows changes to be made to software without risk of losing previous iterations. At any point software can be reverted to a previous version if needed. VC also helps with agile development as deliverables can be easily identified in a VC system.

Git is a standard VC system used in development. Git was chosen as github provides free repositories which can be used to store code. This makes code accessible for anywhere and greatly reduces the risk of it being lost. Github makes git repositories incredibly easy to set up and maintain.

3.6.2 Python Libraries

Various external python libraries were used to facilitate development.

Pandas is a library containing DataFrames which allow easy manipulation of data. This includes labelling variables and calculating contingency tables which is important in the χ^2 test of independence.

NetworkX was used for graphical models, the Graph and Digraph were used and extended to store various graphs used in the algorithms. The built in functionality of greatly helped with the implementation of the PDAG and PAG classes. NetworkX also has functions for finding paths in graphs, this was especially useful for the FCI algorithm.

Finally, Scipy was used to calculate the χ^2 statistic and p-value.

3.7 Refactoring

Refactoring is a very important part of the agile framework and allows a developer to apply newly gained knowledge to old code. Refactoring consists of rewriting functioning code to improve it.

This improvement could be one of many things. Obvious things like speed and memory consumption may be optimised. However, refactoring can also aid in

making code more readable or using a new library that is better suited to the task.

Refactoring may however fail and render previously functional code non-functional. However, this is easily mitigated by string unittesting and version control. Unittesting allows a developer to see whether code that has been refactored still functions in the same way as old code, this will show if refactoring has been unsuccessful. In the case where refactoring fails a developer can either attempt to identify the non-functional area of code and repair/replace it or use the version control system to easily roll back to an existing functional version of the software. These two systems hugely mitigate the risks of refactoring.

4 Implementation

4.1 Skeleton

4.1.1 Indpendence Test

The first sprint consisted of implementing the PC algorithm. This began with implementing a function to determine the skeleton of the graph. To estimate the skeleton a function to determine independence and conditional independence of data was needed. The chi squared tests for independence and conditional independence were chosen. This test will only work on discrete data however other tests could be substituted for other data.

Pandas dataframes were chosen to store data during the algorithm as they provide easy indexing and many useful functions. The crosstab function was especially useful when implementing the chi squared tests as it could automatically compute the contingency tables needed for the tests.

A function prepare_data was implemented to read data from a file into a data frame for processing.

The unconditional independence test was fairly straightforward to implement as it only consisted of computing the contingency tables and then comparing the values in the tables to the sums of the rows and columns in the tables.

The conditional case was more complex, first the conditioning set was combined into a single variable through concatenation of each variable in every data point. This made making a contingency table across all the observed values significantly easier. Next a contingency table across the three variables was needed. Again the cross tab function was used, however calculating the sums of each time a variable takes a particular value was now more complex, this is because the columns for the conditioning set were split between the columns for

the Y variable.

On top of this not all values of Y and Z were observed simultaneously in data so we cant simply iterate through all values of y and z and some up the associated columns as some do not exist. First the columns that exist must be found and then they can be summed over. After this the test is similar to the unconditional however the expected valued will now also account for the value of the conditioning set.

The scipy chi2 function was then used to calculate the chi2 statistic and p-value which is returned by the function.

4.1.2 Variables to be tested

With this function the skeleton can be found. A fully connected networkx Graph is generated and then edges between nodes that show conditional independence are removed.
First unconditional independence tests are performed between each pair of variables and edges removed between those that are found to be independent. The conditional tests based on variables adjacent to the first variable with size of the set increasing by one after all tests have been performed. The algorithm tests $X \perp\!\!\!\perp Y|Z$ and $Y \perp\!\!\!\perp X|Z$, this may seem inefficient as it appears that tests are being repeated. However, the members of the set Z are dependent on the first variable in the statement. This means that the tests must all be performed.

To find the conditioning sets, the combinations function from the python library itertools was used to find all subsets of $adj(x, G)$ y of a certain size.

4.1.3 Order dependence

The members of set the conditioning set Z for the test $X \perp\!\!\!\perp Y|Z$ depend on the tests performed before it. This is due to Z consisting of nodes adjacent X, however the set of odes adjacent to X changes as edges are removed.

In the R implementation an order independent version has been developed. When the size of conditioning sets is incremented, the conditioning sets for all pairs of variables are calculated and stored. However the original order dependent version can still be used.

4.1.4 Separation sets

Edge orientation requires the separation set of each pair of variables. This was stored as a dictionary of all pairs of edges and simply updated with a condition-

ing set if that set was found to condition independence between the pair.

Since all of the algorithms learn a skeleton, the function is a method of the GraphLearner class which is the parent class of the other algorithm classes. The function returns a networkx Graph containing the estimated edges of the skeleton and a dictionary containing the separation set of each pair of variables.

4.2 PC edge orientation

For the pc algorithm edge orientation is relatively simple. However, the pc algorithm outputs a PDAG which contains a mixture of directed and undirected edges, networkx does not contain a data structure to support this kind of graph so one had to be implemented. This turned out to be fairly trivial, using a directed graph behind the scenes in which undirected edges were represented as bidirected edge. Since it's a DAG there cant be any actual bidirected edges so all bidirected edges can simply be considered to be undirected.

When finding the edges orientation, one graph was used to store directed and one used to store undirected. When an edge was directed it was simply transferred from one to the other and once all edges that can be oriented are oriented, the two graphs are combined into a PDAG.

4.2.1 V-Structures

The first part of edge orientation consists of identifying colliders. This is fairly simple as you simply need to find V structures and test if the node that is adjacent to two of the variables is in their separation set. If it is orient the edges toward that node.

However, there may be situations in which the order of the v-structures you test affects the final orientation of the graph. There may be two connected v-structures where orienting one will also orient the other but not in the direction described by the constraints.

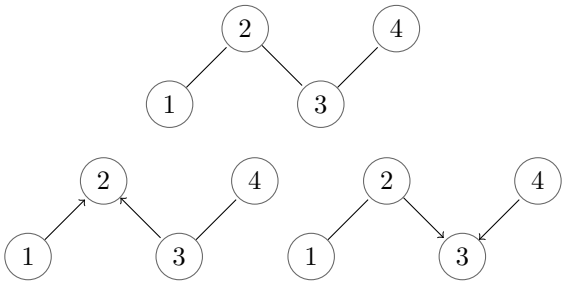
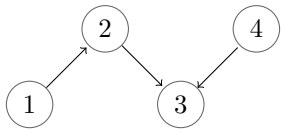


Figure - shows two possible orientations, depending on which v structure is oriented first, in the first case when we arrive at the 2,3,4 triple one of two thing can be done. The edge (3,4) can be left undirected or the new structure can override the old one, as shown in figure - .



In the prexisiting R implementation the second approach of overwriting edges was taken so the same was done in the python implementation.

Finally the other orientatian rules described in the algorithm were followed, these were easy to implement using the networkx graphs as they consisted of testing for undirected and directed edges, then orienting based on those tests. This was repeated until no more edges were oriented, this completed the algorithm

4.2.2 FCI Algorithm

The FCI algorithm was able to use exactly the same code and first round v-structure orientation of the pc algorithm. However after this code unique to the fci algorithm must be developed.

First the final skeleton of the graph must be found. This is done by performing more independence tests, conditioned on the possible d-separating sets of the variables. Finding these sets was made simple by a networkx function to find paths between nodes on a graph.

For the rest of the algorithm the network is a PAG. The data structure used for the PAG was a subclass of the networkx undirected graph. In this graph the direction of an edge was determined by tags stored in a dictionary associated with that edge. This allowed all edge types of a PAG to be stored.

4.2.3 Testing

To ensure that the algorithm was behaving as intended all parts must be thoroughly tested. The python unittest module was used to create a test suite covering as many aspects of the algorithm as possible.

For the chi squared independence test a sanity check consisting of testing whether a variable was independent of itself was used. Obviously the test will result in a p-value of 0 so was very easy to to test. This test was repeated in the case

of a conditioning set of size 1 and size 2 to ensure this did not affect functionality.

Next a pre-existing test from r was used to find results on real data. This data was tested with the python implementation and the results compared. Data from the alarm_10000 and the sia_1000 datasets were used. Many tests with different variables and conditioning sets were performed.

To test the skeleton estimation again the R implementation was used. However, the R version does not use a chi squared test, a wrapper needed to be written to make it compatible with the R skeleton function. With this, skeletons could be generated from data in the same way in python and R and the results compared. The separation sets were also recorded and compared.

Testing the edge orientation for the PC algorithm first started with small sanity checks. For example the collider orientation was tested on networks with 3 nodes and two edges and simple separation sets.

The path checking needed for the pc algorithm was also tested on small networks where paths were linear and some with many branches. It was also tested on a few negative examples.

Finally a skeleton and separation set generated from the alarms data was oriented in both R and python to test the whole edge orientation process as a whole. However due to the order dependence of the edge orientation the implementations produced slightly different results for the same graphs.

This meant that to test the edge orientation section of the pc algorithm, graphs had to be manually oriented using the algorithm and then compared to those estimated by the software.

For the FCI algorithm, the first part of the skeleton estimation is the same as in the PC algorithm which is confirmed to function correctly. However, the final skeleton estimation needs to be tested so comparison testing was done with the R implementation.

Comparison of the edge orientation was performed, however due to the number of edge orientation rules it is likely that only some of them would be applied in a particular run. This means that to fully test the rules they should be tested individually. So the rules were tested on graphs which should be oriented by the rule and that shouldn't be oriented by the rule. The graphs were then tested after rule had been applied to see if the rule functioned correctly.

4.3 Final Structure

The final implementation consisted of four main components. Theχ^2 test for independence, the skeleton estimation, PC edge orientation, and FCI edge orientation.

4.3.1 Independence Test

The χ^2 test was a stand-alone function that can take a data frame along with the names of variables to be tested and a conditioning set. it returns a the χ^2 statistic and a p-value from the test. The p-value represents the probability that the variables are not dependent conditioned on the conditioning set, if this statistic is high enough the variables can be considered independent.

4.3.2 Graph Learners

Each algorithm is implemented as a class, the base class of all algorithms is the graph leaner class. These class takes data and an independence test and learn a graph.

The base class implements the skeleton learning that all the algorithms use. It can also find and orient colliders on that skeleton. Finally, it has a static method to prepare data, this reads data from a file into a dataframe which can be used by the other methods.

The PC algorithm and the FCI algorithms were implemented as subclasses of the Graph learner. The PC algorithm only needed to implement the edge orientation step to complete the algorithm.

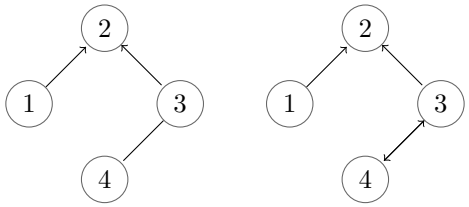
The FCI algorithm had to both implement edge orientation and extend the skeleton estimation of the graph learner.

4.3.3 Graphs

Two new graph classes were implemented, the PDAG and the PAG. The PDAG is a fairly simple extension of networkx's built in Directed graph class in which undirected edges are merely stored as bidirected edges, in PDAGS there are no bidirected edges so ther will be no confusions between bidirected and undirected edges.

This implementation is simple but fairly powerful. Most built in networkkx functions will work in a reasonable manner. For example, the has_edge function will be direction dependent on directed edges but not on undirected edges. Path finding algorithms will also work assuming undirected edges can be traversed in

either direction.

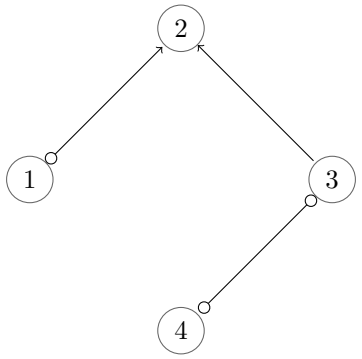


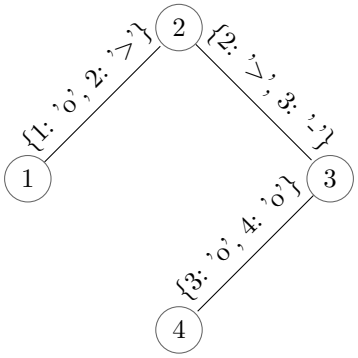
The PAG was more complex to implement due to the increased complexity of the graph. The PAG class is an extension of the undirected graph from networkx. All information about the tags, and therefore the directionality of the edges is stored in an attribute associated with particular edge. The attribute is a dictionary with keys being the nodes at each end of the edge and the values being the tag at that end of the edge.

This implementation meant that mayn new methods needed to be added and many existing methods modified. For example, when adding a new edge the tags of the edge needed to be assigned. methods for testing if edges were directed from u to v (had an arrow head into v) or fully directed from u to v (tail at u, arrowhead at v) were created and used during the FCI algorithm.

The built in methods for finding paths did however still work with this implementation which was very useful for edge orientation.

Currently viewing the graph as a whole is tricky, the built in functions for displaying graphs can not display the edge tags. It may be possible to use the display edge labels function to show the tags as a label on the graph.





5 Analysis

Analysis of this project will mainly consist of comparison to th implementation fo the algorithms in R. It will compare the functionality of the algorithms on data and also the run speed of the code.

5.1 Initial Skeleton Generation

The skeleton generation code has almost identical functionality as the R implementation. Even the order in which edges are tested is the same. However the R implementation has a non-order dependent mode which is not currently implemented in the python version however could be added as an option in the future.

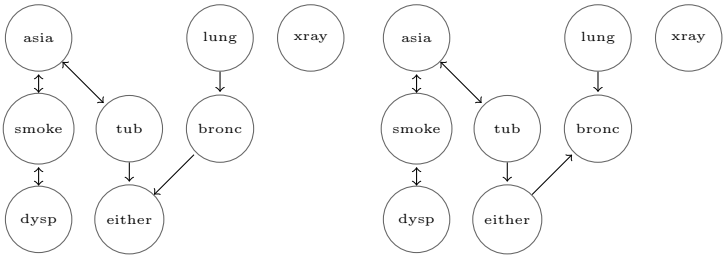
The Alarm_1000, Asia_1000, and Diabetes_1000 datasets found in all produce identical skeletons as seen in Figure _.

The χ^2 test for independence was not implemented in the pcalg library in R. Other independence test for discrete data, such ad the G^2 test, were implemented. However, the bnlearn library has an implementation of the χ^2 test to use for comparison.

Both tests return identical results on all tested data as seen in

5.2 PC Algorithm Edge Alignment

5.2.1 Asia_10000



5.2.2 Diabetes_1000

5.2.3 Differences

There are some differences in the edge orientation of the different implementations. For example, in the orient of asia_1000 network, the edge either-bronc is oriented in opposite directions by the two implementations. This is due to the order in which the colliders were oriented in the graph. In the R implementation the triple lung, bronc, either is oriented followed by the triple tub, either bronc. This causes the edge either-bronc to be overwritten and reoriented. In the python implementation this is done in the opposite order resulting in a different network.

Neither order is more correct than the other however this orientation is used to estimate the final skeleton of the FCI algorithm so differences in the order can have a large knock on effect in the PAGs estimated by this algorithm.

5.3 FCI Algorithm Final Skeleton

5.4 FCI Algorithm Edge Alignment

5.4.1 Asia_10000

5.4.2 Diabetes_1000

5.4.3 Differences

5.5 Run Speed

5.5.1 Method

5.5.2 Measurements

5.5.3 Algorithms

5.6 Skeleton Generation

5.6.1 Independence Tests

The independence test was the main bottleneck of the algorithms, and was significantly slower than the R implementation. This is because the R implementation actually runs C code behind the scenes, c code is compiled rather than interpreted like python and R. This makes the code much faster.

However the independence test in the algorithm is a parameter and thus the current slow test could be replaced by a faster version which calls C code like the R implementation.

6 Conclusion

6.1 Further Work

The most obvious extension to this project would be to implement the FCI algorithm described in the literature review.

Another improvement would be to increase the speeed of the conditional independence tests. One approach could be to implement the independence test in C and interface with it using Cython. C programs are much faster than those written in python so there would almost definitely be an improvement.

Other independence tests could also be written. These tests could be for the same kind of data as the χ^2 or it could be a test for other data for example continuous data.

References

[1] Thomas Verma and Judea Pearl. Causal networks: Semantics and expressiveness. In *Machine Intelligence and Pattern Recognition*, volume 9, pages 69–76. Elsevier, 1990.

[2] Jiji Zhang. Causal reasoning with ancestral graphs. *Journal of Machine Learning Research*, 9(Jul):1437–1474, 2008.

[3] Judea Pearl. Causality: models, reasoning, and inference. *Econometric Theory*, 19(675-685):46, 2003.

[4] Judea Pearl. Causal inference in statistics: An overview. *Statist. Surv.*, 3:96–146, 2009.

[5] Richard Scheines. An introduction to causal inference.

[6] Diego Colombo, Marloes H Maathuis, Markus Kalisch, and Thomas S Richardson. Learning high-dimensional directed acyclic graphs with latent and selection variables. *The Annals of Statistics*, pages 294–321, 2012.

[7] Peter Spirtes and Clark Glymour. An algorithm for fast recovery of sparse causal graphs. *Social science computer review*, 9(1):62–72, 1991.

[8] Jiji Zhang. On the completeness of orientation rules for causal discovery in the presence of latent confounders and selection bias. *Artificial Intelligence*, 172(16):1873 – 1896, 2008.