Reproduction Information

Transborder electrical interties do not stimulate development of Canadian hydroelectric resources

Amir Mortazavigazar[1,2,*], Mark E. Borsuk[3], and Ryan S.D. Calder[1,2,3,4,5]

[1]Department of Population Health Sciences, Virginia Tech, Blacksburg, VA, 24061, USA
[2]Global Change Center, Virginia Tech, Blacksburg, VA, 24061, USA
[3]Department of Civil and Environmental Engineering, Duke University, Durham, NC, 27708, USA
[4]Faculty of Health Sciences, Virginia Tech, Roanoke, VA, 24016, USA
[5]Department of Civil and Environmental Engineering, Virginia Tech, Blacksburg, VA, 24061, USA
[*]Corresponding author: `amirgazar@vt.edu`

Contents

1 Configuration and Setup

This document presents a comprehensive guide outlining the sequential process required to execute and replicate the causal inference modeling methodology presented in our article. Additionally, we supplied the pseudocode overview of code functionality in Section 3.

- To execute the supplied code refer to Section 2 of this document.
- To replicate this code refer to Sections 3 to 13 of this document.

1.1 Hardware and Operating System

We utilized a MacBook Pro with an Apple M1 Pro chip, featuring an 8-core CPU and 16 GB of memory. The startup disk is the Macintosh HD. The system operates on macOS 13.2.1 (22D68) `Ventura`.

1.2 Software Versions and Dependencies

We used R version 4.3.1 (2023-06-16) `Beagle Scouts` for the x86_64-apple-darwin20 platform. Furthermore, we used the RStudio environment, version 2023.06.2+561 to execute the our code. Table 1 below shows packages and versions utilized in this code.

Table 1: Package Version

Package	Version
Rgraphviz	2.44.0
bnlearn	4.8.3
gRain	1.3.13
visNetwork	2.1.2
ggplot2	3.4.2
zoo	1.8.12
scales	1.2.1
gridExtra	2.3
dplyr	1.1.2
MASS	7.3.60
svglite	2.1.1
tidyverse	2.0.0

1.3 Install Time

Installation of RStudio and R is contingent upon the specific hardware and operating system in use. This also holds true for the packages employed within the code. However, the packages used in this code typically take a few minutes to install.

1.4 Run Time

The run time on our operating system for this code is approximately 15 seconds.

2 Full code execution instructions

Following the instruction below, you can execute our code for the 5-year lag period and generate DAGs and conditional dependency graphs. This code is in turn broken up into an annotated and justified workflow in Sections 5 to 13. The expected outputs for this code are visualized DAGs presented in the `viewer` tab in RStudio and conditional dependency plots are saved as `.svg` files in the `Hydro EIA Code` folder. Additional results are printed in the `console` in RStudio.

1. Download the `Hydro_EIA_Code.zip` file available in this link..
2. Unzip the `Hydro_EIA_Code.zip` file.
3. Open the `Hydro_EIA_Code.R` in RStudio.

4. Install any required packages and load them, refer to Section 5.1.
5. Set the working directory to the `Hydro EIA Code` folder. See Section 5.2.
6. Execute the code.
7. The conditional dependency figures will be saved automatically in the `Hydro EIA Code` folder. The DAGs will be displayed in the `viewer`, all other results are displayed in the `console`.

3 Pseudocode Overview

Here we provide the pseudocode overview of our algorithm.

Algorithm 1 Pseudocode Overview of Code Functionality

```
 1: START
 2: INITIALIZE the R environment:
 3:     a. Install and load required packages.
 4:     b. Set the working directory.
 5: DEFINE required functions:
 6:     a. DAG visualizer.
 7:     b. Box-Cox transformer.
 8:     c. Goodness of fit test.
 9:     d. D-separation test.
10: PREPARE the data:
11:     a. Generate delta and lags.
12:     b. Create averaged and lagged variables.
13:     c. Ensure data is numeric.
14:     d. Check for Gaussian distribution.
15:     e. Transform non-Gaussian variables.
16:     f. Discretize variables.
17:     g. Create the final data-frame.
18: CREATE a blacklist.
19: VISUALIZE the hypothesized DAG.
20: CONSTRUCT score-based DAGs.
21: PLOT conditional dependency results.
22: TEST goodness of fit for each node.
23: PERFORM d-separation analysis.
24: END
```

4 Algorithm Overview

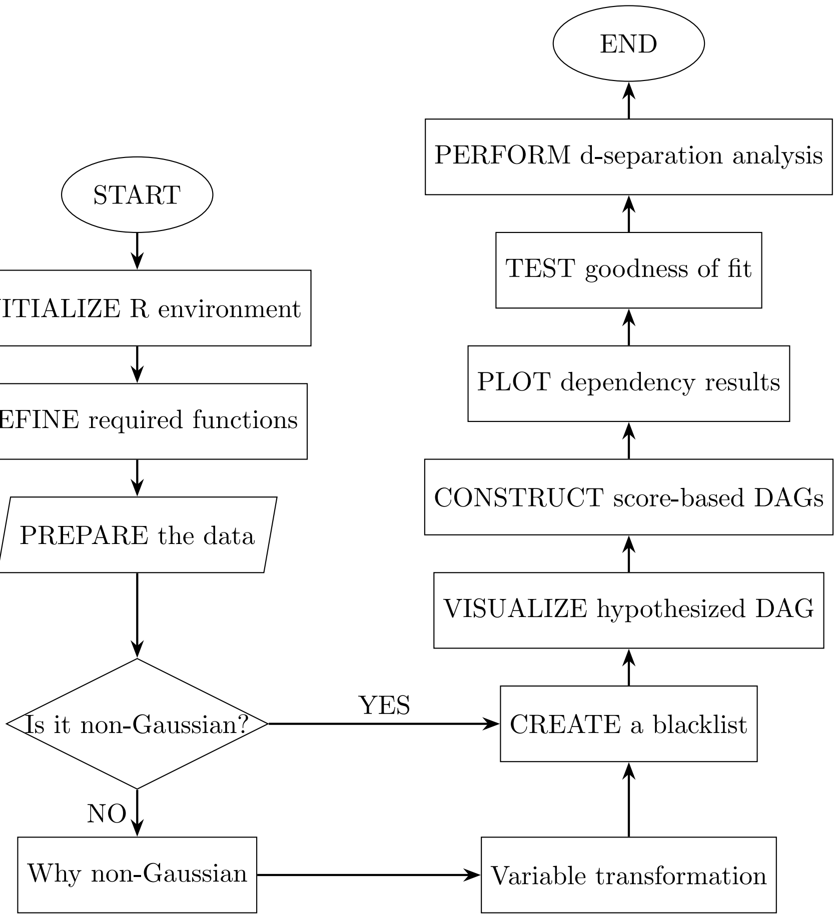

Figure 1: Algorithm to construct and evaluate BN models.

5 Preparing the R Environment

5.1 Install and Load the Required Packages

We use the following packages throughout the code. The core package that must be installed is the `bnlearn` package[1].

```
# Install Rgraphviz from Bioconductor
if (!requireNamespace("BiocManager", quietly = TRUE))
    install.packages("BiocManager")
BiocManager::install("Rgraphviz")

# List of other packages to install
packages <- c("bnlearn", "gRain", "visNetwork", "ggplot2",
              "zoo", "scales", "gridExtra", "dplyr", "MASS", "svglite", "tidyverse")

# Install packages
install.packages(packages)

# Load all the required packages
invisible(lapply(c("Rgraphviz", "bnlearn", "gRain", "visNetwork", "ggplot2",
```

[1]Scutari, M., Silander, T., and Ness, R. (2023). Bayesian Network Structure Learning, Parameter Learning and Inference (4.8.3) [R]. Available: `https://www.bnlearn.com/`

```
                    "stats", "zoo", "scales", "gridExtra", "dplyr", "MASS", "svglite", ...
                        "tidyverse"), library, character.only = TRUE))
```

5.2 Set Working Directory

Set the working directory and read in the data. Make sure to adjust the directory path to match your setup.

```
# Set Working Directory (change for your setup)
setwd("/Users/Documents/Hydro EIA Code")
rm(list=ls())
# Read the data-set
hydro.data <- read.csv("hydro_var_aug23.csv")
```

6 Required Functions

We have created the following functions that will assist in creation and evaluation of our causal directed acyclic graphs (DAGs).

6.1 DAG Visualizer

Visualizes and returns DAGs created by the `bnlearn` package.

```
plot.network <- function(structure, ht = "400px", title){

  # Unique nodes from the arcs of the structure are identified.
  nodes.uniq <- unique(c(structure$arcs[,1], structure$arcs[,2]))

  # A data frame for nodes is created with attributes like id, label, color, and shadow.
  nodes <- data.frame(id = nodes.uniq,
                      label = nodes.uniq,
                      color = "maroon",
                      shadow = TRUE)

  # A data frame for edges is created with attributes like source, target, arrow ...
      direction, and other visual properties.
  edges <- data.frame(from = structure$arcs[,1],
                      to = structure$arcs[,2],
                      arrows = "to",
                      smooth = TRUE,
                      shadow = TRUE,
                      color = "black")

  # The network is visualized using the visNetwork function and returned.
  return(visNetwork(nodes, edges, height = ht, width = "100%"))
}
```

6.2 Box-Cox Transformer

Evaluates the data-set and checks for normality (using Shapiro-Wilk test), transforms the non-Gaussian variables using Box-Cox transformation. Re-evaluates the transformed variables with the Shapiro-Wilk test and checks for normality. Returns the results.

```
transform_and_test <- function(df, non_gaussian_vars){

  # Lists to store variables that remain non-Gaussian after transformation and those that ...
      were successfully transformed.
  still_non_gaussian <- vector("list")
  transformed_vars <- vector("list")
  df_new = df

  # Each non-Gaussian variable is processed.
  for (var in non_gaussian_vars) {

    # The minimum value of the variable is determined.
    min_value <- min(df[[var]], na.rm = TRUE)

    # If the minimum value is less than or equal to zero, a constant is added to make it ...
        positive.
    if (min_value ≤ 0) {
      constant <- abs(min_value) + 1
      df[[var]] <- df[[var]] + constant
    }

    # The Box-Cox transformation parameter (lambda) is estimated.
    bc <- boxcox(df[[var]] ¬ 1, plotit = FALSE)
    lambda <- bc$x[which.max(bc$y)]

    # Depending on the value of lambda, the variable is transformed.
    if(abs(lambda) ≤ 1e-5){
      transformed_var <- log(df[[var]])
    } else {
      transformed_var <- (df[[var]]^lambda - 1) / lambda
    }

    # The transformed variable is tested for normality using the Shapiro-Wilk test.
    shapiro_test <- shapiro.test(transformed_var)

    # Based on the p-value, the variable is categorized as still non-Gaussian or ...
        successfully transformed.
    if (shapiro_test$p.value < 0.05) {
      still_non_gaussian <- c(still_non_gaussian, var)
    } else {
      df_new[[var]] <- transformed_var
      transformed_vars <- c(transformed_vars, var)
    }
  }

  # The modified data frame, list of still non-Gaussian variables, and list of transformed ...
      variables are returned.
  list(df = df_new, still_non_gaussian = still_non_gaussian, transformed = transformed_vars)
}
```

6.3 Goodness of Fit Test

Evaluates each variable's goodness of fit for the DAGs produced by the **bnlearn** package.

Function to compute the R^2 (r-squared) metric for continuous variables, representing the proportion of variance in the dependent variable that's predictable from the independent variable.

```
evaluate_fit_continuous <- function(actual, predicted) {

  # An empty list for metrics is initialized.
  metrics <- list()
  # The r-squared metric is computed and stored in the metrics list.
  metrics$rsquared <- 1 - sum((predicted - actual)^2) / sum((actual - mean(actual))^2)
  # The metrics list is returned.
  return(metrics)
}
```

Function to compute the accuracy metric for discrete variables, defined as the ratio of correctly predicted values to the total number of values.

```
evaluate_fit_discrete <- function(actual, predicted) {

  # An empty list for metrics is initialized.
  metrics <- list()
  # The accuracy metric is computed and stored in the metrics list.
  metrics$accuracy <- sum(actual == predicted) / length(actual)
  # The metrics list is returned.
  return(metrics)
}
```

6.4 D-Separation Test

Uses conditional dependence test to evaluate relationships between independent nodes in the DAG produced by the **bnlearn** package while factoring in the available data.

We have created the `dsep.dag` function that can evaluate d-separation for any given node pair. This function uses the optimized DAG results to identify the following for each node pair and then calculates conditional independence: 1. Parents, 2. Neighbors (i.e., parents and children for each node) and 3. Markov-Blanket (i.e., parents, children and parents of children for each node). Furthermore, this function uses data to evaluate d-separation in addition to the results of DAG discovery. This combines the functionalities of the `ci.test` and `dsep` functions available in the `bnlearn` package. Where, `ci.test` exclusively utilizes data, while `dsep` solely employs DAGs.

6.4.1 Function Usage

The `dsep.dag` function is used as follows:

```
dsep.dag(x, data, z)
```

6.4.2 Parameters

The `dsep.dag` function accepts the following parameters:

- `x`: an object of class `bn`
- `data`: a data frame containing the variables in the model
- `z`: a list, where each element is a character vector representing a pair of node labels

Available conditioning sets (that are printed automatically) are:

1. **parents**: set of causal variables for each node
2. **neighbors**: set of causal and response variables for each node
3. **markov_blanket**: set of causal and response variables including causal variables of responses for each node

6.4.3 Return Value

The `dsep.dag` function returns the following for each node pair:

- `Results for parents`: a character string
- `Results for neighbors`: a character string
- `Results for markov_blanket`: a character string

The character string can yield one of the following values, representing the outcome of the d-separation analysis:

1. **Conditionally Independent**: The variables are independent given the observed variables.
2. **Potential Missing Link**: There might be a missing link between the variables.
3. **Uncertain - Further Analysis Needed**: The relationship between the variables is uncertain, necessitating further analysis.
4. **NA**: The node pair is either not available in the DAG, or they are connected, implying conditional dependence.

6.4.4 Underpinning Functions

To construct and utilize the `dsep.dag` function, we implemented the following functions sequentially:

Function to check if a directed arc exists between two nodes in a given set of arcs. This eliminates manually checking the arcs in an existing DAG.

```
arc_exists <- function(from, to, existing_arcs) {
  # The existence of the arc is checked and the result is returned.
  return(any(existing_arcs[existing_arcs[, "from"] == from, "to"] == to))
}
```

Function to perform d-separation tests on all non-adjacent pairs of nodes in a given DAG using the dataset provided.

```
perform_dsep_tests <- function(dag, data) {
  # Existing arcs in the DAG are extracted.
  existing_arcs <- arcs(dag)

  # An internal function to check for arc existence is defined.
  arc_exists <- function(from, to, existing_arcs) {
    return(any(existing_arcs[existing_arcs[, "from"] == from, "to"] == to))
  }

  # A list for non-adjacent node pairs is initialized.
  non_adj_pairs <- list()

  # Non-adjacent node pairs are identified.
  for (node1 in nodes(dag)) {
    for (node2 in nodes(dag)) {
      if (!arc_exists(node1, node2, existing_arcs) && !arc_exists(node2, node1, ...
          existing_arcs) && node1 != node2) {
        non_adj_pairs <- append(non_adj_pairs, list(c(node1, node2)))
      }
    }
  }

  # A list for test results is initialized.
  results_parents <- list()
  results_nbr <- list()
  results_mb <- list()

  # D-separation tests are performed for each non-adjacent pair.
  for (pair in non_adj_pairs) {
    # Conditioning on parents of a node
    conditioning_set <- setdiff(unique(c(bnlearn::parents(dag, pair[1]), ...
        bnlearn::parents(dag, pair[2]))), pair)  # Combined Markov blanket excluding x and y
    test <- ci.test(pair[1], pair[2], conditioning_set, data = data)
    results_parents[[paste0(pair[1], "_", pair[2])]] <- test$p.value
    # Conditioning on immediate neighbors of a node
    conditioning_set <- setdiff(unique(c(nbr(dag, pair[1]), nbr(dag, pair[2]))), pair)  # ...
        Combined Markov blanket excluding x and y
    test <- ci.test(pair[1], pair[2], conditioning_set, data = data)
    results_nbr[[paste0(pair[1], "_", pair[2])]] <- test$p.value
    # Conditioning on Markov Blanket of a node
    conditioning_set <- setdiff(unique(c(mb(dag, pair[1]), mb(dag, pair[2]))), pair)  # ...
        Combined Markov blanket excluding x and y
    test <- ci.test(pair[1], pair[2], conditioning_set, data = data)
    results_mb[[paste0(pair[1], "_", pair[2])]] <- test$p.value
  }

  return(list(parents = results_parents, neighbors = results_nbr, markov_blanket = ...
      results_mb))

}
```

Function to categorize the p-values from d-separation tests into three categories: "Conditionally Independent", "Potential Missing Link", and "Uncertain - Further Analysis Needed".

```r
interpret_dsep_pvalues <- function(pvalues, threshold_low = 0.05, threshold_high = 0.95) {
  # P-values are categorized based on predefined thresholds.
  categories <- sapply(pvalues, function(p) {
    if (p > threshold_high) {
      return("Conditionally Independent")
    } else if (p < threshold_low) {
      return("Potential Missing Link")
    } else {
      return("Uncertain - Further Analysis Needed")
    }
  })
  return(categories)
}
```

Function to organize and print the interpret_dsep_pvalues function results.

```r
interpret_print <- function(node1, node2, interpretations_list) {
  # Constructing the key string from the node names
  key <- paste0(node1, "_", node2)

  interpretations <- lapply(interpretations_list, function(interpretation) {
    interpretation[key]
  })
  names(interpretations) <- c("parents", "neighbors", "markov_blanket")
  # Printing the results
  cat("From", node1, "To", node2, ":\n")
  for (name in names(interpretations)) {
    cat("Results for", name, ":\n")
    cat(interpretations[[name]], "\n\n")  # Directly prints the value without the key
  }
}
```

Function that wraps all of the previous functions into one concise function, i.e., the `dsep.dag` function.

```r
dsep.dag <- function(dag, data, node_pairs) {
  # Performing D-separation tests
  dsep_results <- perform_dsep_tests(dag, data)

  # Translating the results
  interpretations_list <- lapply(dsep_results, interpret_dsep_pvalues)

  # Initializing a list to hold the interpreted results for each node pair
  interpreted_results <- list()

  for (node_pair in node_pairs) {
    key <- paste0(node_pair[1], "_", node_pair[2])
    interpret_print(node_pair[1], node_pair[2], interpretations_list)
    interpreted_results[[key]] <- lapply(interpretations_list, function(interpretation) ...
        interpretation[key])
  }

  # Returning the list of interpreted results
  return(interpreted_results)
}
```

7 Data Preparation

Modify the variables as needed, we introduced lags and deltas to ensure we capture the temporal difference of causality for our variables. Refer to the main document for further information. Note that in this section we only show the code for some variables transformations. Refer to the main document for further details for each transformation.

7.1 Generating delta and lags

We manually introduced deltas and lags into our variables based on the hypothesized model.

```
{
  # New intertie capacity in every year
  hydro.data$INTERTIE_new = NA
  for(i in 2:nrow(hydro.data)){
    hydro.data$INTERTIE_new[i] = hydro.data$INTERTIE[i] -
      hydro.data$INTERTIE[i-1]
  }
  # Sum of new intertie capacity in preceding 5 years
  hydro.data$INTERTIE_5y = NA
  for(i in 5:nrow(hydro.data)){
    hydro.data$INTERTIE_5y[i] = sum(hydro.data$INTERTIE_new[(i-4):i],na.rm=T)
  }
  # Sum of new intertie capacity in preceding 5 years lagged by 5 years
  hydro.data$INTERTIE_5y_lag_5y = NA
  for(i in 10:nrow(hydro.data)){
    hydro.data$INTERTIE_5y_lag_5y[i] = hydro.data$INTERTIE_5y[i-5]
  }
}
```

7.2 Creating averaged and lagged variables

We used loops to generate new variables that represent 5-year averages and their respective lags.

```
lag_periods <- c(5)
new_vars <- c("INTERTIE", "INSTALLED", "DEMAND_QC", "DEMAND_US", "INVESTMENT", "EXPORTS", ...
    "PRICE")

for (var in new_vars) {
  var_new <- paste0(var, "_new")

  for (lag_period in lag_periods) {
    # moving average
    var_avg <- paste0(var_new, "_avg_", lag_period, "y")
    hydro.data[[var_avg]] <- zoo::rollapplyr(hydro.data[[var_new]], width = lag_period, ...
        FUN = mean, fill = NA)

    # lagged average
    var_lag_avg <- paste0(var_avg, "_lag_", lag_period, "y")
    hydro.data[[var_lag_avg]] <- dplyr::lag(hydro.data[[var_avg]], lag_period)
  }
}
```

7.3 Creating a subset for averaged and lagged variables

We subset our data-set to only retain 5-year averaged and lagged variables.

```
vars.exclude.5y = c(1:8,grep("_new$",colnames(hydro.data)))
hydro.data.subset.5y =
  hydro.data[,setdiff(1:ncol(hydro.data),
                      vars.exclude.5y)]
```

7.4 Ensuring data-set is numeric

It is critical to ensure that the data-set is numeric before proceeding to perform any further analysis.

```
for(i in 1:ncol(hydro.data.subset.5y)){
  hydro.data.subset.5y[,i] = as.numeric(hydro.data.subset.5y[,i])
}
```

7.5 Creating a new data-frame

After preparing our data-set we create a new data-frame that excludes rows with missing values. It must be noted that introducing lags and deltas will result in some information loss, therefore selecting an optimum lag period is imperative.

```
# Create new dataframe
df.5y = hydro.data.subset.5y
```

```

# Creating new data frames with minimum number of rows cut off
lag.cols.5y = grep("lag",colnames(df.5y))
df.5y.no.lags = df.5y[,setdiff(1:ncol(df.5y),
                               lag.cols.5y)]
df.5y.rows.to.cut = which(apply(df.5y,1,function(x) sum(is.na(x))>0))
df.5y.no.lags.rows.to.cut = which(apply(df.5y.no.lags,1,function(x) sum(is.na(x))>0))
df.5y.with.lags.no.NA = df.5y[setdiff(1:nrow(df.5y),
                                      df.5y.rows.to.cut),]
```

7.6 Check for Gaussian distribution

We perform the Shapiro-Wilk test to check for normality across our variables. Bayesian models we use in this analysis require normality to ensure an accurate estimation. Variables that don't pass the Shapiro-Wilk test a p-value less than the specified significance level (0.05) are considered non-Gaussian and are stored in a list.

```
significance_level <- 0.05
non_gaussian_5y_lag <- vector("list")
for (var in colnames(df.5y.with.lags.no.NA)) {
  # Shapiro-Wilk Test
  shapiro_test <- shapiro.test(df.5y.with.lags.no.NA[[var]])
  print(paste("Shapiro-Wilk Test for", var, "- p-value:", shapiro_test$p.value))

  if (shapiro_test$p.value < significance_level) {
    non_gaussian_5y_lag <- c(non_gaussian_5y_lag, var)
  }
}
```

7.7 Transforming non-Gaussian variables

We transform the variables that are non-Gaussian, obtained from the previous step. We use the `'transform_and_test'` function created in Section 6.2 to transform the variables using Box-Cox transformation. If the variables are still non-Gaussian after transformation the function reverts them to their original state.

```
result_5y_lag <- transform_and_test(df.5y.with.lags.no.NA, non_gaussian_5y_lag)
df.5y.with.lags.no.NA <- result_5y_lag$df
still_non_gaussian_5y_lag <- result_5y_lag$still_non_gaussian
```

7.8 Discretizing variables

Variables that cannot be transformed using the Box-Cox transformation are then discretized into categories based on their values and histograms.

```
df.5y.with.lags.no.NA$EXPORTS_new_avg_5y <- cut(df.5y.with.lags.no.NA$EXPORTS_new_avg_5y, ...
    breaks = c(min(df.5y.with.lags.no.NA$EXPORTS_new_avg_5y,na.rm=T), 0, ...
    max(df.5y.with.lags.no.NA$EXPORTS_new_avg_5y,na.rm=T)), labels = c("negative", ...
    "positive"), include.lowest = TRUE, ordered_result = TRUE)
```

7.9 Creating the final data-frame

We drop the variables that are identical after transformations. Finally we create the final data-frame by subsetting the current data-frame.

```
# Drop the variables
var.drop = c("INSTALLED_new_avg_5y", "INTERTIE_new_avg_5y","INSTALLED_new_avg_5y_lag_5y", ...
    "INTERTIE_new_avg_5y_lag_5y", "INSTALLED_lag_5y", "INTERTIE_lag_5y")
# Subset the dataset
df.5y.with.lags.no.NA =
  df.5y.with.lags.no.NA[,setdiff(1:ncol(df.5y.with.lags.no.NA),
                                 match(var.drop,colnames(df.5y.with.lags.no.NA)))]

# Selecting columns of interest
selected.columns <- c('INSTALLED_5y_lag_5y', 'INSTALLED_5y', ...
    'DEMAND_QC_new_avg_5y_lag_5y', 'INVESTMENT_5y', 'PRICE_5y_lag_5y',
                        'INTERTIE_5y_lag_5y', 'INTERTIE_5y', 'DEMAND_US_new_avg_5y_lag_5y', ...
                             'PRICE_5y',
                        'EXPORTS_new_avg_5y_lag_5y', 'EXPORTS_5y', 'DEMAND_QC_5y', ...
                            'DEMAND_US_5y')

# Subsetting the dataframe and creating the final dataframe
df.expert.5y <- df.5y.with.lags.no.NA[, selected.columns]
```

8 Creating a Blacklist

We have to set the parameters and boundaries for our Bayesian model. We do this by creating an allowable list of arcs derived from our hypothesized ”expert” model. Using a for-loop we then convert this allowable list to a blacklist.

```
# The allow list is initialized with specific variable pairs.
allow.list.expert =
  data.frame(matrix(c(
    # Various variable pairs are listed here.
    "INVESTMENT_5y","INSTALLED_5y",
    ...
    "DEMAND_US_5y","PRICE_5y"),
    ncol = 2,byrow=TRUE))

# Column names for the allow list are assigned.
colnames(allow.list.expert) = c("From","To")

# The black list is initialized with a placeholder value.
black.list.expert = NA

# For each pair of variables in the final data-frame, a check is performed.
# If the pair is not found in the allow list, it is added to the black list.
for(i in 1:ncol(df.expert.5y)){
  for(j in 1:ncol(df.expert.5y)){
    from.test = colnames(df.expert.5y)[i]
    to.test = colnames(df.expert.5y)[j]

    if(length(which(allow.list.expert$From==from.test&
                    allow.list.expert$To==to.test))==0){
      black.list.expert =
        rbind(black.list.expert,c(from.test,to.test))
    }
  }
}

# Column names for the black list are assigned.
colnames(black.list.expert) = c("From","To")

# The placeholder value in the black list is removed.
black.list.expert = black.list.expert[2:nrow(black.list.expert),]
```

9 Visualizing the Hypothesized DAG

We construct the hypothesized ”expert” DAG using the ’**model2network**’ function from the **bnlearn** package. We visualize this DAG using ’**plot.network**’ function that we created in Section 6.1. Figure 2 shows the visualized expert DAG.

```
# A DAG is constructed using expert knowledge.
dag.expert.5y <- model2network("[INSTALLED_5y_lag_5y][DEMAND_QC_new_avg_5y_lag_5y]
[PRICE_5y_lag_5y][INTERTIE_5y_lag_5y][DEMAND_US_new_avg_5y_lag_5y][EXPORTS_new_avg_5y_lag_5y]
[DEMAND_QC_5y][DEMAND_US_5y]
[INSTALLED_5y|DEMAND_QC_new_avg_5y_lag_5y:INVESTMENT_5y:PRICE_5y_lag_5y:INTERTIE_5y_lag_5y]
[INTERTIE_5y|INSTALLED_5y_lag_5y:INVESTMENT_5y:DEMAND_US_new_avg_5y_lag_5y:PRICE_5y_lag_5y]
[INVESTMENT_5y|DEMAND_QC_new_avg_5y_lag_5y:EXPORTS_new_avg_5y_lag_5y]
[EXPORTS_5y|INTERTIE_5y:INSTALLED_5y:PRICE_5y]
[PRICE_5y|DEMAND_QC_5y:DEMAND_US_5y]")

# The constructed DAG is visualized with a specified height.
plot.network(dag.expert.5y, ht = "600px")
```

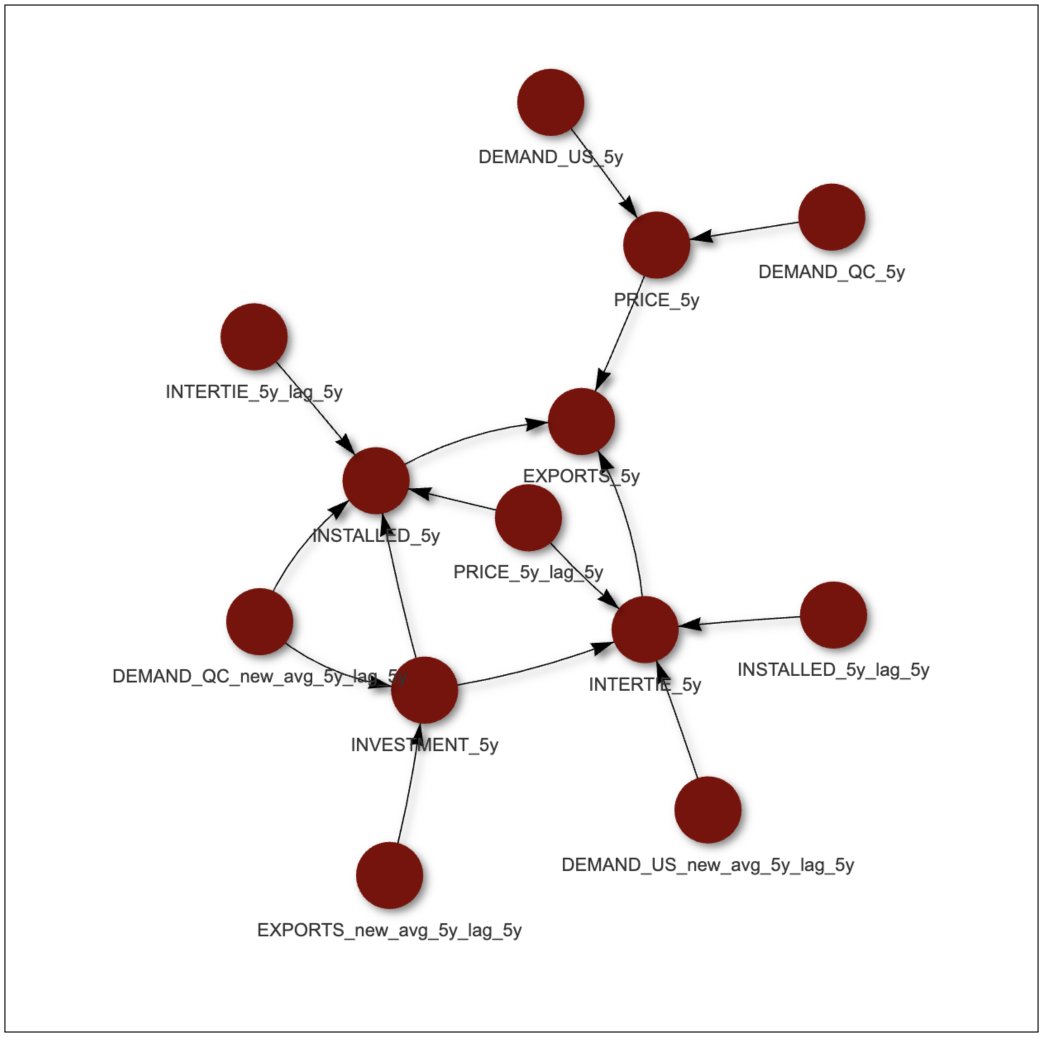

Figure 2: Hypothesized DAG visualization.

10 Constructing Score-Based DAGs

We construct DAGs using the score-based hill-climb (HC) algorithm. W use the `'hc'` and `'bn.fit'` functions from the `bnlearn` package. We visualize this DAG using `'plot.network'` and `'graphviz.chart'` functions. We use three different scoring functions; 1. `'loglik-cg'`, 2. `'aic-cg` and 3. `'bic-cg'`.

```
# DAG created using the loglik-cg score function and HC algorithm
dag.expert.5y.emp <- hc(df.expert.5y, score = "loglik-cg", blacklist = black.list.expert, ...
    debug = FALSE)
par(mar=c(1,1,1,1))
# Fitting the model
model.expert.5y.emp = bn.fit(dag.expert.5y.emp, df.expert.5y)
#Visualizing model's conditional probabilities using the graphviz.chart
graphviz.chart(model.expert.5y.emp,  type = "barprob", grid = TRUE, bar.col = "darkgreen",
                strip.bg = "lightskyblue")
dev.off()
# Network visualized using plot.network
plot.network(dag.expert.5y.emp, ht = "600px")

# DAG created using the aic-cg score function and HC algorithm
dag.expert.5y.emp.aic <- hc(df.expert.5y, score = "aic-cg", blacklist = black.list.expert)
plot.network(dag.expert.5y.emp.aic, ht = "600px")
par(mar=c(1,1,1,1))
```

```r
model.expert.5y.emp.aic = bn.fit(dag.expert.5y.emp.aic, df.expert.5y)
graphviz.chart(model.expert.5y.emp.aic,  type = "barprob", grid = TRUE, bar.col = "darkgreen",
               strip.bg = "lightskyblue")
dev.off()

# DAG created using the bic-cg score function and HC algorithm
dag.expert.5y.emp.bic <- hc(df.expert.5y, score = "bic-cg", blacklist = black.list.expert)
plot.network(dag.expert.5y.emp.bic, ht = "600px")
par(mar=c(1,1,1,1))
model.expert.5y.emp.bic = bn.fit(dag.expert.5y.emp.bic, df.expert.5y)
graphviz.chart(model.expert.5y.emp.bic,  type = "barprob", grid = TRUE, bar.col = "darkgreen",
               strip.bg = "lightskyblue")
dev.off()
```

11 Plotting Conditional Dependency Results

We used the conditional probability results from the DAG model to evaluate the relationship between various nodes. We used visualization to plot these dependencies. This was achieved by creating a separate R code called `Graph_Generator.R` located in the same directory as the main code. This code is then recalled using the `source` function.
It must be noted that the type of each plot is dependant on the type of each node and the number of its parents. Therefore, here we only present one node's results as an example. Refer to the supplemental information document for further details for each node. Figure 3 shows the results for this plot.

11.1 Running the `Graph_Generator.R` Code

We use the following code to recall the `Graph_Generator.R` code from the directory. This will run that code in the background.

```r
source("Graph_Generator.R")
```

11.2 Snippet of `Graph_Generator.R` Code

Here is an snippet of `Graph_Generator.R` code where graphs are defined for nodes of interest and they're saved in the `.svg` format automatically when this code is called.

```r
# 5. Conditional density: EXPORTS_5y | INSTALLED_5y + INTERTIE_5y + PRICE_5y
# Makind predictions for the node EXPORTS_5y using 'predict' function from the 'bnlearn' ...
    package.
df.expert.5y$EXPORTS_5y_pred = predict(model.expert.5y.emp, node = "EXPORTS_5y", data = ...
    df.expert.5y, method = "bayes-lw")
p<- ggplot(df.expert.5y, aes(x = PRICE_5y, y = INSTALLED_5y, size = EXPORTS_5y_pred, color ...
    = as.factor(INTERTIE_5y))) +
  geom_point() +
  labs(x = "PRICE",
       y = "INSTALLED",
       size = "Predicted EXPORTS",
       color = "INTERTIE") +
  scale_color_manual(values = palette)
ggsave(filename = "fig_8.svg", plot = p, device = "svg")
```

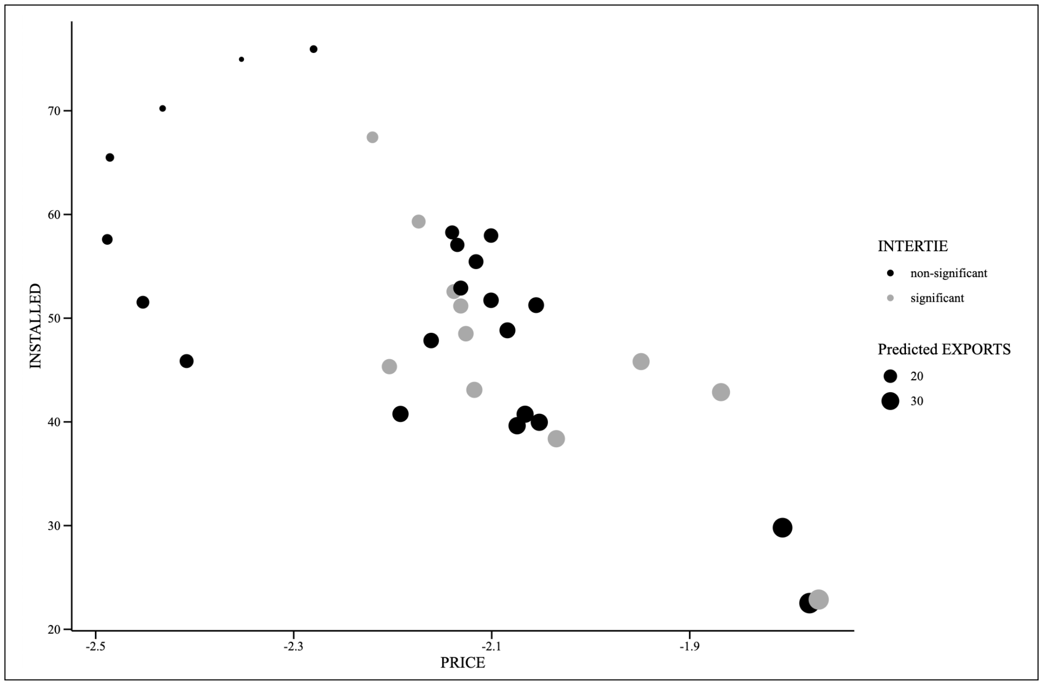

Figure 3: 5-year model's conditional dependency results for the exports node.

12 Goodness of Fit for Each Node

Here we use the `'evaluate_fit_discrete'` and `'evaluate_fit_continuous'` functions created in Section 6.3 to evaluate the goodness of fit for nodes with parent(s). We use the `predict()` function from the `bnlearn` package. Additionally we utilize Monte Carlo posterior inference method using the `bayes-lw` with 5000 parameters.

```
# identify discreet and continiuous variables
discrete_vars <- c("INTERTIE_5y", "INVESTMENT_5y")
continuous_vars <- setdiff(colnames(df.expert.5y), c(discrete_vars, value = TRUE))
# Create a blank results list
results_loglik <- list()

# Looping over each node to generate its prediction using the 'predict' function from the ...
    'bnlearn' package.
for (var in colnames(df.expert.5y)) {
  if (!grepl("_pred$", var)) {
    pred_column <- paste(var, "pred", sep = "_")
    df.expert.5y[[pred_column]] <- predict(model.expert.5y.emp, node = var, data = ...
        df.expert.5y, method = "bayes-lw", n = 5000)
    actual_values <- df.expert.5y[[var]]
    predicted_values <- df.expert.5y[[pred_column]]

    # Calculate the goodness of fit for different type of variables using the functions we ...
        created previously
    if (var %in% continuous_vars) {
      results_loglik[[var]] <- evaluate_fit_continuous(actual_values, predicted_values)
    } else if (var %in% discrete_vars) {
      results_loglik[[var]] <- evaluate_fit_discrete(actual_values, predicted_values)
    }
  }
}
```

13 D-Separation Analysis

Here we use the `'arc_exists'`, `'perform_dsep_tests'` and `'interpret_dsep_pvalues'` functions created in Section 6.4 to evaluate the conditional dependence of nodes of interest if needed. This step helps resolve any discrepancies or nuances that we might witness in different models.

```r
# Defining the node_peirs list to identify nodes of interest where we want to perfom ...
    d-separation. Note that this list contains node in a "from", "to" format.
different_edges <- compare(dag.expert.5y.emp, dag.expert.5y, arcs = TRUE)
node_pairs <- different_edges$fp
print(node_pairs)
node_pairs <- lapply(seq_len(nrow(node_pairs)), function(i) as.character(node_pairs[i, ]))

# Using the dsep.dag function to calculate conditional dependancy for each pair
dsep_log <- dsep.dag(dag.expert.5y.emp, df.expert.5y, node_pairs)
```