Universidad Internacional de Valencia

Máster en Inteligencia Artificial

Causal Discovery Unit Testing

Titulación:
Máster en Inteligencia Artificial

Curso académico:
2019-2020

Lugar de residencia, mes y año:
Barcelona, Noviembre 2020

Alumno/a:
Adrien Felipe

D.N.I:
ooooooooo

Director:
Dr. Gherardo Varando

Convocatoria:
Primera

Orientación:
Investigación

Créditos:
12 ECTS

4 de noviembre de 2020

UNIVERSIDAD INTERNACIONAL DE VALENCIA

Resumen

Máster en Inteligencia Artificial

Causal Discovery Unit Testing

por Adrien FELIPE

Una necesidad fundamental presente en múltiples dominios, y no necesariamente científicos, es poder explicar y predecir un sistema a partir de las relaciones de causa y efecto que dominan sus propiedades o eventos. Para ello, la exploración causal o **Causal Discovery** propone técnicas y algoritmos con el propósito de descubrir dichas relaciones, a partir de observaciones del sistema. Desafortunadamente no resulta trivial dominar el funcionamiento preciso de cada algoritmo ni tampoco escoger la estrategia de búsqueda más apropiada según la problemática. Sería conveniente disponer de una herramienta que permita valorar y comparar diversos algoritmos de exploración aplicado específicamente al caso particular. Partiendo de conjuntos de datos independientes y representativos de cada caso concreto de relación causal, se podría valorar qué tipos de relaciones un algoritmo es realmente capaz de encontrar, y cuáles no.

Inversamente, durante la integración de un algoritmo de exploración causal, o durante su mantenimiento, es importante poder medir su buen funcionamiento. En el desarrollo de software es habitual disponer de pruebas automáticas de código, que garantizan el cumplimiento de los requisitos de la aplicación. Resultaría ventajoso poder aplicar los mismos patrones de comprobación, sobre los algoritmos de exploración causal.

Este trabajo se centrará en desarrollar un framework basado en *Python* que permita crear fácilmente conjuntos de datos relacionales, sobre los cuales ejecutar y valorar uno o varios algoritmos de exploración causal. Adicionalmente, se definirá los tipos de relaciones básicas representativos del abanico completo de las posibles relaciones causales. De esta forma se obtendrá un punto inicial completo de valoración, pero a la vez disponiendo de la flexibilidad de adaptarse a las necesidades específicas de cada proyecto. Mientras no esté dentro del alcance del TFM, la intención es poder integrar en un futuro esta aplicación dentro de un proceso de integración continua como capa de tests unitarios o **Unit Testing**.

Este trabajo no se basa en un proyecto anterior de su director el Dr. Gherardo VARANDO.

Índice

Índice de figuras

Agradecimientos

A través de estas líneas quiero expresar mi más sincero agradecimiento a todas las personas que hicieron posible este máster en inteligencia artificial.

Agradezco muy especialmente a mi director de TFM el Dr. Gherardo VARANDO, por su tiempo y paciencia, con quien he podido no solo realizar este trabajo pero además profundizar en el ámbito de exploración causal que desconocía por completo.

Quiero agradecer a todos los profesores de este máster quienes han sido claves en transmitirnos la base de conocimientos necesaria y quienes me han inspirado en más de una ocasión.

A mi compañero de máster Enrique NAVARRO por su interés y acompañamiento a lo largo del desarrollo de este curso a pesar de sus condiciones virtuales.

Finalmente, una gratitud particular a mi querida prometida la Dra. Marta RAJKIEWICZ por su soporte y comprensión durante todo este tiempo dedicado día tras día a este curso y trabajo final.

Introducción

1.1 Estado del Arte

Existen múltiples dominios dentro de los cuales es necesario poder explicar relaciones de causa y efecto [22, 24, 18]. Hasta la democratización de herramientas de computación y de aprendizaje automático [19], la metodología estándar de encontrar relaciones dentro de un sistema se ha basado a lo largo de los siglos en experimentos controlados [13, 6], modificando una a una características y observando los resultados consecuentes, determinando así empíricamente las relaciones que lo dominan. No obstante, esta técnica resulta ser frecuentemente inviable o incluso imposible [16, 25], y por ese motivo la **exploración causal** aporta una solución al estar basada en el análisis de los datos puramente observacionales. Esta se define por algoritmos que exploran los datos con técnicas estadísticas, construyendo una representación de las diversas relaciones causales.

Dicha representación puede ser puramente probabilística, a partir de modelos de grafo no dirigido o UGM (Undirected Graphical Models) - también conocidos como **Campos aleatorios de Markov** - que describen **correlaciones** entre variables y se limitan a una determinación de independencia [34]. Sin embargo no siempre es suficiente, y resulta necesario incorporar un conocimiento estructural de las relaciones con modelos de grafo dirigido o DGM (Directed Graphical Models) - también llamados **Redes bayesianas** - que describen relaciones de **causalidad**.

Formalizar relaciones entre variables definidas y consistentes, de forma a representar, estimar y probar un modelo causal, tiene una base teórica conocida como modelo de ecuaciones estructurales o **Structural Equation Models** (SEM) [21], que a partir de los **modelos de grafo dirigido y no dirigido** permite una mayor interpretación y validación [27, 19].

1.1.1 Modelo de ecuaciones estructurales

Un *SEM* puede considerarse como un sistema de relaciones que proporcionan consistencia e interpretación a fenómenos reales. Las causas y los efectos son generados por **funciones** que calculan el valor de algunas variables a partir de otras variables. Estas funciones no han de ser estrictamente matemáticas, sino más bien elementos procedurales lógicos que devuelven un valor de salida según sus valores de entrada [19, p. 83].

Un elemento importante de los *SEM* es permitir introducir **variables latentes**, que no estarían presentes directamente a partir de los datos observacionales, y poder así describir relaciones escondidas.

J. Peters define las ecuaciones estructurales como una colección de asignaciones funcionales X_i, y una distribución conjunta de eventos independientes de ruido N_i, tal que:

$$X_i := f_i(\mathbf{PA_i}, N_i)$$

donde $\mathbf{PA_i} \subseteq \{X\} \setminus \{X_i\}$ es llamado padres de X_i o **causas directas** [19, p. 84]. La asignación de las relaciones ha de ser acíclica, es decir que ningún X_i puede ser causa de él mismo, ni directa o indirectamente. Una matiz a este último punto, es en el caso de relaciones no instantáneas o con desfase temporal, llamadas **series temporales** [19, 10 Time Series], donde la relación de una variable a ella misma tal que $X_i^t := f_i(X_i^{t-n}, N_i) \ \forall n > 0$ sí es acíclica y es permitido.

La formalización de estas asignaciones funcionales permite generar grafos dirigidos acíclicos descriptivos de las relaciones de causa y efecto entre las variables del sistema. La Figura 1.1 ilustra un ejemplo de ecuaciones estructurales y su grafo correspondiente. En este caso la variable X_1 es considerada la causa padre al ser función únicamente de un ruido aleatorio N_1. La asignación del resto de variables X_2, X_3 y X_4 es función de alguna otra variable además de un ruido aleatorio independiente, los cuales son considerados efectos.

$$\begin{aligned} X_1 &:= f_1(N_1) \\ X_2 &:= f_2(X_1, N_2) \\ X_3 &:= f_3(X_2, N_3) \\ X_4 &:= f_4(X_3, X_1, N_4) \end{aligned}$$

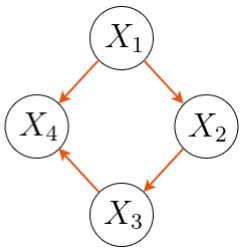

FIGURA 1.1: Un ejemplo de *SEM* con su asignación funcional (izquierda) y el grafo dirigido acíclico correspondiente (derecha).

En la práctica los *SEMs* se componen de tres etapas:

1. Definir las variables del sistema, sean latentes o no.
2. Construir los grafos estructurales con una formalización de las relaciones internas.
3. Validar el modelo a partir de técnicas de exploración causal.

Asimismo, a partir de un modelo formalizado de ecuaciones estructurales, es posible validar modelos de aprendizaje de estructura causal. Estos modelos, a la inversa de los *SEMs* han de descubrir el grafo estructural sin formalización previa. Para ello existen principalmente tres métodos: los basados en restricciones, en puntuación o en asimetrías.

1.1.2 Métodos basados en restricciones

Esta técnica consiste primero en crear un grafo no dirigido completo de las posibles relaciones, e ir orientando y eliminando los arcos progresivamente [19, p. 143]. El algoritmo más común y también uno de los más antiguos es el **algoritmo PC** [26], del cual declinan múltiples variantes

como MAGs, PAGs, o FCI a modo de ejemplo.

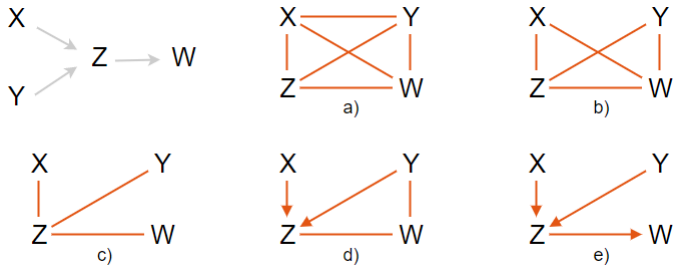

FIGURA 1.2: Representación del funcionamiento del algoritmo PC.

Se puede observar en la Figura 1.2 cómo el algoritmo PC empieza con todas las variables X, Y, Z y W conectadas con arcos no dirigidos, los cuales elimina paso a paso a la vez que los orienta, hasta encontrar las relaciones finales.

1.1.3 Métodos basados en puntuación

Los algoritmos basados en puntuación intentan encontrar la estructura causal maximizando la probabilidad del grafo a partir de una función de valoración previamente definida. A la inversa de los métodos basados en restricción, parten de un grafo vacío [31]. El algoritmo principal de esta familia es el **algoritmo GES** [5] del cual también declinan los algoritmos FGES, SGES, SE-GES entre otros.

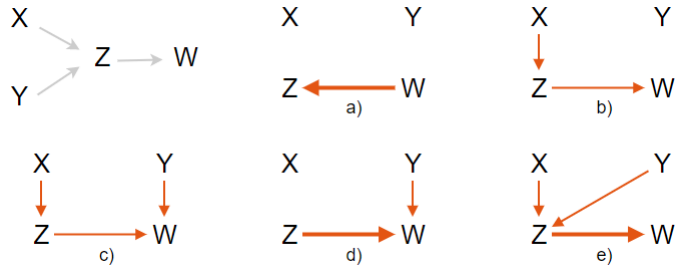

FIGURA 1.3: Representación del funcionamiento del algoritmo GES.

En la Figura 1.3 se ilustra cómo el algoritmo GES parte de un grafo vacío y crea los arcos progresivamente con más o menos probabilidad según la valoración de la función de puntuación hasta tener convergencia.

1.1.4 Métodos basados en asimetrías

Estos algoritmos se definen como no-gaussianos o no-lineales y su objetivo es estimar los parámetros de una distribución de probabilidad no clásica para describir las relaciones causales

[12]. El **algoritmo LiNGAM** [23] es el más habitual dentro de esta familia, de lo cual declinan los DirectLiNGAM, ICA-LiNGAM.

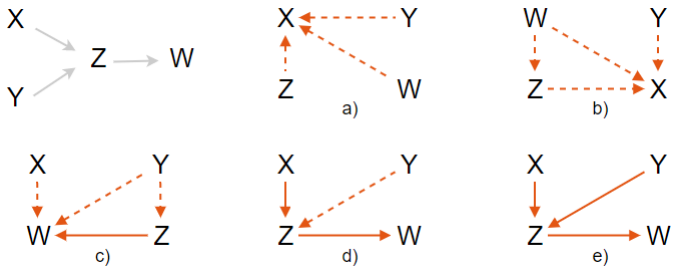

FIGURA 1.4: Representación del funcionamiento del algoritmo LiNGAM.

La Figura 1.4 ilustra una representación de la construcción de un grafo acíclico a partir de permutaciones de variables del algoritmo LiNGAM.

1.2 Motivación

La variedad de algoritmos dedicados a la exploración causal es amplia y la capacidad de cada uno a encontrar la estructura causal es muy dependiente del conjunto de datos y de su tipo de relaciones. Existen publicaciones cuyo objetivo es comparar diversos algoritmos [25, 31, 9], y si resultan de gran ayuda como punto de partida, los datos y relaciones usados para hacer los benchmarks no necesariamente cumplen con los mismos criterios y especificidades de cualquier proyecto, o no comparan exactamente los algoritmos deseados y de la forma deseada, centrándose generalmente en los algoritmos más comunes.

Existen herramientas o plataformas como CauseMe [14] o Causality Workbench [11] diseñadas justamente con el fin de ejecutar y comparar propuestas de algoritmos de exploración. Estos entornos proponen múltiples conjuntos de datos diferentes con diversos tipos de relaciones, pero cuyas relaciones no son explícitamente conocidas o descritas, asimismo están más orientados a la evaluación y competición más que a ser usados dentro de un proceso automatizado de validación de código. Dentro del desarrollo de software es habitual, para garantizar la calidad de la aplicación, disponer de procesos automáticos de comprobación de su correcto funcionamiento así como del cumplimiento de los requisitos de dominio [7], conocido como Integración continua o **Continuous Integration** (CI) [8]. Es posible imaginar una herramienta de business intelligence, un video juego, o cualquier aplicación donde sea necesario integrar y ampliar algoritmos de exploración causal a la vez de garantizar su buen funcionamiento de forma automática.

Aplicar este proceso de validación a algoritmos de exploración causal requiere disponer de conjuntos de datos cuyas relaciones son conocidas, sobre los cuales comprobar que los grafos

encontrados por los algoritmos correspondan con los valores esperados. Esta descripción justamente encaja con la definición de modelos de ecuaciones estructurales descrita en 1.1.1, pero no solo en formalizar relaciones causales para validar un algoritmo de exploración causal, sino permitiendo adicionalmente las combinaciones necesarias para garantizar su calidad y su perennidad. Esto se puede asimilar a lo conocido como pruebas unitarias o **Unit testing** [28, 3] aplicado a **Causal discovery**.

1.3 Objetivo

Partiendo del concepto de ecuaciones estructurales, este TFM pretende desarrollar un mini framework dentro del cual sea posible ejecutar exploraciones causales como si fueran tests unitarios y asimismo poder ser integrado dentro de un proceso de integración continua. Para ello la aplicación tendrá que permitir generar dinámicamente conjuntos de datos a partir de configuraciones o funciones, ser capaz de conocer las relaciones causales correspondientes, y permitir ejecutar exploraciones para ser valoradas. Con este fin el trabajo se divide en tres bloques:

1) Definir los tipos de relaciones causales
Es necesario determinar los diferentes tipos de relaciones causales para poder implementar un modelo de ecuaciones estructurales que pretenda permitir generar cualquier tipo de relación. Adicionalmente la intención de este proyecto es que contenga por defecto conjuntos unitarios correspondientes a cada tipo de relación.

2) Desarrollar un generador de conjuntos de datos
Existen librerías para generar datasets e incluso las propias librerías de causal discovery permiten generar sintéticamente conjuntos de datos, pero siempre muy enfocados al tipo de exploración de la librería, permitiendo un abanico limitado de casos. Se pretende con esta aplicación poder generar cualquier tipo de ecuación estructural y disponer del grafo correspondiente. Esto será responsabilidad de lo nombrado aquí *Generador*, el cual tendrá que permitir generar conjuntos sintéticos de cada tipo de relación, así como conjuntos más completos que puedan representar casos reales.

3) Implementar un comparador / valorador de algoritmos
El objetivo final de este proyecto es poder fácilmente ejecutar exploraciones causales relevantes con los algoritmos deseados y medir sus resultados, de forma a poder integrarse en un proceso de control de calidad automático, lo cual llamaremos el *Comparador*. Aunque no forme parte de este TFM integrarse dentro de un proceso de integración continua, sí se buscará tener una salida de valoraciones explotable para ello.

Preparación y entorno

2.1 Docker

Una problemática habitual a la hora de usar una librería, es instalar sus dependencias, así como las versiones correctas sin que estas entren en conflicto con lo que esté ya instalado. Este proyecto está desarrollado en Python y por tanto depende de varios paquetes de este lenguaje. Anaconda, pip y otros gestores de paquetes permiten instalar fácilmente las dependencias de los paquetes de python así como también configurar entornos virtuales donde incluso disponer de versiones diferentes de python según los requisitos de la aplicación.
Resulta muy práctico siempre y cuando el proyecto no requiera de librerías no vinculadas a python, como R por ejemplo, o incluso módulos y extensiones adicionales, configuraciones especiales, etc, ya que todo ha de estar instalado y configurado en la máquina donde se ejecute la aplicación. Y aunque este proyecto no esté diseñado para ser desplegado en un entorno de producción, sí que debería ser autosuficiente y poder ejecutarse sin necesidad de instalar manualmente y en la propia máquina paquetes y librerías. Para esto mismo existe Docker, el cual nos permite construir y levantar lo equivalente a una máquina virtual a partir de un fichero de configuración, desde el cual podemos instalar todas las librerías y dependencias de forma automatizada y multi-plataforma sin afectar la máquina anfitriona.
De esta forma, tras descargar el código de este proyecto, solo bastaría con construir la imagen de docker con el comando `docker build` y disponer así de todo lo necesario instalado y listo para su uso, sin siquiera tener python instalado en el ordenador.
Consultar el apéndice A para el fichero de configuración de la imagen de docker.

2.2 Makefile

Docker facilita la portabilidad y aislamiento del entorno, pero ejecutar sus comandos puede resultar tedioso, sobre todo que levantar el contenedor con una imagen de docker no siempre es suficiente. Suele ser necesario aplicar cambios al contenedor una vez levantado, como por ejemplo activar servicios, o hacer gestiones que dependan del estado actual de ejecución, desconocido a la hora de crear la imagen. Se podría perfectamente crear bash scripts para gestionarlo, pero la utilidad make simplifica la gestión y lectura de dichos comandos. Podemos por ejemplo iniciar el contenedor con un comando personalizado `make start` el cual por detrás ejecuta secuencialmente comandos complejos de docker necesarios para levantar y configurar el contenedor. No es el propósito inicial de make, al este ser de facilitar la compilación de código fuente, pero *make* es ante todo también una herramienta de automatización [15] y resulta mucho más simple de mantener que un bash script para lo mismo.

El apéndice B detalla el contenido del fichero de configuración de *make* donde se pueden apreciar los comandos implementados.

2.3 Ssh

El tener un entorno de ejecución virtualizado o dockerizado significa que el intérprete de Python también lo es, y por tanto que no está directamente accesible desde la máquina donde se trabaje. Jupyter como la mayoría de entornos de desarrollo (IDE) permiten ejecutar intérpretes remotos mediante una conexión SSH. Por esto mismo el comando `make start` descrito en 2.2 ejecuta los comandos necesarios para levantar el servicio ssh en el contenedor de docker, crea un acceso con el mismo id de usuario que el que esté activo en el ordenador, y evita así problemas de permisos entre los ficheros creados dentro y fuera del contenedor.

2.4 Tests

Este proyecto se orienta a disponer de pruebas automáticas de algoritmos de Causal discovery, pero no quita que el propio código ha de estar bajo prueba y garantizar así su buen funcionamiento. Por esto mismo por cada una de las clases principales desarrolladoras para esta aplicación, se han creado adicionalmente los tests correspondientes y los diferentes casos de uso relevantes. Finalmente, el test coverage [2] indica un 80% de líneas de código bajo prueba, y más del 90% de los ficheros de la clase principal. Aprovechando el makefile, se puede ejecutar la batería completa de tests simplemente con `make test`.

2.5 Github

El código está subido a github de forma a poder ser compartido así como recibir feedback o propuestas para ampliarlo: AdrienFelipe/Causal-discovery-Unit-testing.

Desarrollo del proyecto

3.1 Tipos de relaciones causales

Las relaciones entre variables causales pueden ser de múltiples tipos, así como ser combinaciones de varios tipos. Aún así, es posible reducirlas a patrones unitarios de relaciones [29, 30] descritos en las secciones a continuación.

3.1.1 Causalidad directa

Se define como causalidad directa el patrón donde la causa tiene un vínculo directo con el efecto, ilustrado en la Figura 3.1 a). Considerando la causa C, el efecto E, y un ruido aleatorio cualquiera N, se puede representar por la siguiente ecuación:

$$E := f(C) + N$$

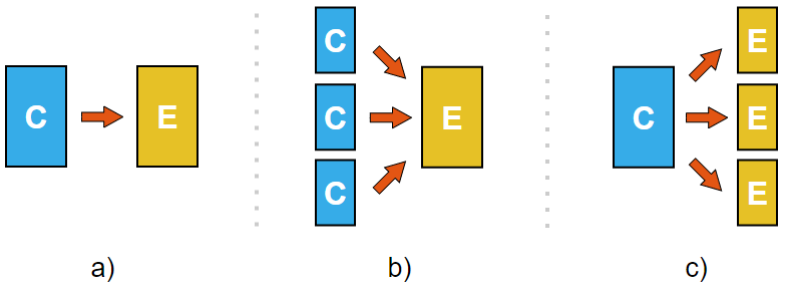

FIGURA 3.1: Representación de causalidad directa y múltiple.

3.1.2 Causalidad múltiple

En el caso de causalidad múltiple, un efecto puede tener múltiples causas, o una causa tener múltiples efectos. Las Figuras 3.1 b) y 3.1 c) ilustran este caso. Podemos escribir el caso b) como el efecto E siendo el resultado de las causas C_1 y C_2 y del ruido N tal que:

$$E := f_1(C_1) + f_2(C_2) + N$$

O inversamente los efectos E_1 y E_2 siendo resultados de la causa C tal que:

$$\begin{cases} E_1 := f_1(C) + N_1 \\ E_2 := f_2(C) + N_2 \end{cases}$$

3.1.3 Causalidad en cadena

También llamado causalidad dominó y representado en la Figura 3.2 d), este patrón corresponde a una relación encadenada de causas y efectos donde las variables interiores son simultáneamente causa y efecto, pero sin que se forme un bucle, es decir sin que un efecto pueda ser causa de una de sus causas. Lo podemos ejemplificar con la siguiente ecuación estructural:

$$\begin{cases} E_1 := f_1(C) + N_1 \\ E_2 := f_2(E_1) + N_2 \\ E_3 := f_3(E_2) + N_3 \end{cases}$$

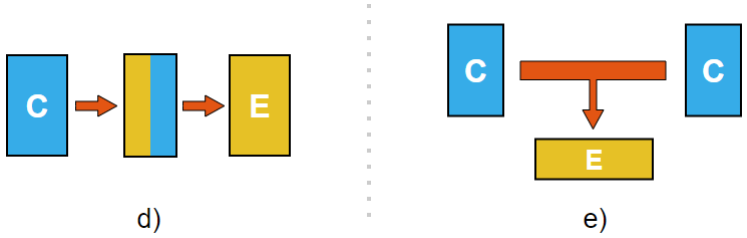

FIGURA 3.2: Representación de causalidad en cadena y relacional.

3.1.4 Causalidad relacional

La causalidad relacional puede asociarse a una causalidad de causa múltiple, pero se distingue en que las causas funcionan en relación de forma que mientras esta se mantenga el efecto no se ve afectado [20]. Ilustrado en la Figura 3.2 e), este patrón puede ser interpretado como el resultado de una única función de múltiples causas:

$$E := f(C_1, C_2) + N$$

3.1.5 Causalidad circular

La Figura 3.3 f) ilustra una causalidad circular donde la causa inicial es también un efecto de la cadena, creando así un bucle. Se puede simbolizar con el sistema de ecuaciones siguiente:

$$\begin{cases} E_1 := f_1(\mathbf{E_0}) + N_1 \\ E_2 := f_2(E_1) + N_2 \\ \mathbf{E_0} := f_3(E_2) + N_3 \end{cases}$$

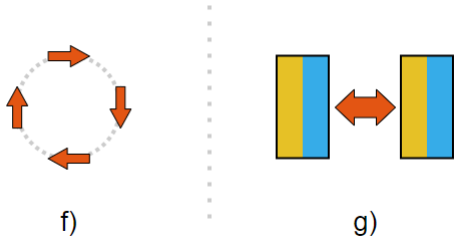

FIGURA 3.3: Representación de causalidad circular y mutua.

3.1.6 Causalidad mutua

Ilustrado en la Figura 3.3 g), este caso corresponde a dos variables que se retroalimentan, siendo una causa de la otra e inversamente. Se podría considerar un caso especial de causalidad circular teniendo un único nivel. Una representación sería la siguiente ecuación:

$$\begin{cases} E := f_1(C) + N_1 \\ C := f_2(E) + N_2 \end{cases}$$

TABLA 3.1: Generación simple de un conjunto de datos.

E1	E2	E3
0	7.975399	0.955
1	0.081354	0.002418
0	6.987423	0.48124

3.2 Generador de conjuntos de datos

3.2.1 Descripción

Se define aquí como *generador* una clase Python responsable de permitir configurar y generar conjuntos de datos relacionales. Es una implementación del modelo de ecuación estructural con el objetivo de poder definir las columnas o variables deseadas de un dataset, y relacionarlas entre ellas con la mayor flexibilidad, tanto directamente como con desfase temporal. Se describe a continuación el funcionamiento de la clase `Generator` y cómo configurar la generación de un conjunto de datos.

3.2.2 Eventos

Para configurar un conjunto de datos a partir del generador es necesario añadirle explícitamente las variables o **eventos**. Estos eventos son los responsables de calcular el valor de su variable correspondiente en cada iteración de generación de una muestra del conjunto de datos. Pueden ser cualquier objeto python siempre que implementen la interfaz `EventInterface`.

Para facilitar la configuración de los conjuntos se han creado varios tipos de eventos de base, como generar un valor discreto aleatorio desde una lista, o un valor continuo desde una distribución gaussiana, o en particular un evento **función** el cual permite crear las relaciones entre variables a partir de cualquier función en python. Cada uno de los eventos disponibles está encapsulado por comodidad en su método correspondiente de la clase generador como por ejemplo `add_categorical` o `add_uniform`. Se ilustra a continuación un caso simple de conjunto de datos con una variable categórica booleana, una variable uniforme aleatoria y por último una función personalizada `custom_function` correspondiente al efecto de las dos primeras variables:

```
generator = Generator() \
        .add_categorical() \
        .add_uniform() \
        .add_function(custom_function)
```

A partir del código anterior es posible generar un `DataFrame` de *pandas* con el método `generator.generate()`. La tabla 3.1 ilustra un ejemplo de valores obtenidos con este mínimo de configuración.

Por defecto las columnas del `DataFrame` son nombradas E_n, pero conviene poder renombrarlas si es necesario, así como también definir la precisión o número de decimales de cada

TABLA 3.2: Conjunto de datos con variables y valores nombrados.

Tiempo	Viento	Temperatura
Cubierto	1	23.6
Lluvioso	67	5.6
Soleado	7	27.6

variable. Podemos completar el ejemplo anterior haciendo por consiguiente más específico el conjunto generado, como ilustrado en la tabla 3.2, con el código a continuación:

```
generator = Generator() \
    .add_categorical(['Soleado', 'Cubierto', 'Lluvioso'], label='Tiempo') \
    .add_uniform(min=0, max=120, label='Viento', round=0) \
    .add_function(temperature_function, label='Temperatura', round=1)
```

3.2.3 Relaciones o evento Función

Para poder crear las relaciones necesarias entre variables se ha definido un evento `Function` que también implementa la interfaz EventInterface. Este evento requiere asociar una función de python definida por el usuario para poder generar el valor de la variable en cada muestra. El interés principal de este evento es que al calcular un valor de muestra por su función asociada, a esta se le inyecta como argumento el resto de todas las variables, así como todos los valores calculados anteriormente, es decir el histórico del conjunto gestionado por la clase `History`. De esta forma se pueden crear relaciones múltiples entre las variables de una misma muestra, pero también relaciones entre muestras creando el desfase temporal necesario para modelos de tipo **Time Series** descrito a continuación en 3.2.4.

El siguiente código ilustra la generación de un conjunto de datos que dispondrá de 3 variables, la última siendo simplemente la suma de las dos primeras a través de la función `custom_function`.

```
def custom_function(history: History):
        return history.get_event(1) + history.get_event(2)

generator = Generator() \
        .add_uniform() \
        .add_uniform() \
        .add_function(custom_function)
```

El método `get_event` de la clase History dispone de un método alias e permitiendo hacer el código más compacto. El mismo ejemplo anterior, pero más compacto con además el uso de una función anónima.

```
generator = Generator() \
        ...
        .add_function(lambda h: h.e(1) + h.e(2))
```

3.2.4 Time series y fecha de evento

Un caso muy habitual en Causal Discovery es explorar datos cuyas relaciones constan de un desfase temporal, conocido como **Time Series** [19, p. 198]. Con el fin de poder crear este tipo de conjuntos, el generador asigna una fecha a cada muestra del dataset, la cual no es más que una variable adicional como el resto de eventos (implementa la interfaz EventInterface). Adicionalmente puede ser parametrizado el valor de tiempo/fecha inicial de la primera muestra, así como la duración del paso entre cada muestra definido en días, horas, minutos o segundos.

Con el código a continuación la primera muestra tendría fecha el 02/01/2020 a las 20:10, y el resto de muestras incrementadas de 5 minutos consecutivamente:

```
generator = Generator() \
        .set_time('2010-01-02 20:10', step='5m')
```

El tiempo entre cada muestra puede no ser constante o más bien ser impreciso. Para ello junto con la definición del step entre muestras es posible definir una precisión correspondiente al rango dentro del cual se generará aleatoriamente la siguiente muestra.

El código a continuación generaría muestras cada hora +/- 10 minutos:

```
generator = Generator() \
        .set_time('2010-10-20 20:00', step='1h', precision='20m')
```

Fecha de muestra

El orden en el cual se añade un evento al generador define su posición, la cual empieza en 1 e incrementa con cada evento añadido. De esta forma se accede al valor de cada variable por su posición: `history.get_event(1)` o más explícitamente `history.get_event(position=1)`. En el caso de la variable de tiempo, esta es añadida expresamente en posición 0 de forma a poder acceder a su valor con `history.get_event(0)`.

Adicionalmente, se han creado los métodos `get_datetime` y `get_timestamp` (o su alias `t`), los cuales no hacen más que encapsular `history.get_event(0)`, para devolver respectivamente un objeto *datetime* y un *unix timestamp* correspondiente al tiempo/fecha de la muestra actual.

Relaciones temporales

Finalmente para crear una relación temporal, el mismo método `get_event` permite acceder a cualquier muestra previamente generada indicando el número de muestras hacia atrás, con el argumento `delay`. Por ejemplo `history.get_event(3, delay=2)` leería el valor de la variable en posición 3, pero 2 muestras antes. Adicionalmente es posible extraer valores por rangos temporales, para buscar si un valor ha superado un umbral en X tiempo por ejemplo. Para ello `history.get_range(3, '10m')` devolvería todos los valores de la variable 3 en los 10 últimos minutos partiendo de la fecha de la muestra actual.

3.2.5 Datos secuenciales

Un caso especial de conjuntos de datos temporales son los casos de lectura de datos tipo sensor o registro donde sólo se dispone del valor de una única variable en cada muestra. Para ello el generador permite ser activado en modo secuencial y cada muestra dispondrá únicamente del valor de una de las variables y su fecha asociada. Adicionalmente los eventos disponen de una propiedad de probabilidad de selección en cada iteración de muestra, por defecto igual a 1 pero la cual conviene definir para estos casos de conjuntos secuenciales.

Podemos reproducir un conjunto de datos tipo log de servidor con el ejemplo a continuación:

```
generator = Generator(sequential=True) \
        .add_categorical(
                ['notice', 'warning', 'critical'],
                probability=0.2
        ).add_function(server_down, probability=0.9)
```

El código anterior define una probabilidad de 20% de seleccionar la variable categórica en cada iteración de muestra, así como una probabilidad del 90% de seleccionar la variable de función. Y adicionalmente, el evento función `server_down` es libre de decidir si devolver un valor nulo en cual caso corresponde a que la variable no ha sido seleccionada dando lugar a seleccionar uno de los siguientes eventos.

Mismamente, todas las etiquetas de una variable categórica disponen por defecto de la misma probabilidad, pero si fuera necesario se les puede definir probabilidades diferentes.

```
generator = Generator(sequential=True) \
        .add_categorical(
                ['notice', 'warning', 'critical'],
                probability=0.2,
                weights=[0.7, 0.25, 0.05]
        )
```

Teniendo así `notice` una probabilidad de 70% de ser elegida cuando el evento es ejecutado (él mismo con 20% de probabilidad), `warning` un 25% y `critical` un 5%.

3.2.6 Variables latentes

No siempre todas las variables y relaciones son observables directamente desde el conjunto de datos. Para poder reproducir estos casos de variables escondidas, los eventos disponen del parámetro `shadow=True`, el cual permite eliminar dicha variable del conjunto final. De esta forma los eventos latentes coexisten de la misma manera que el resto de eventos durante la generación de las muestras, pero no son directamente observables desde el conjunto de datos generado.

TABLA 3.3: Conjunto con la variable E_2 escondida.

E1	E3	E4
3	1	10
9	6	87
1	2	3

El código a continuación ejemplifica un caso donde la variable en posición 2 es efecto de la primera variable, y a la vez causa de la variable en posición 4, pero siendo una variable latente.

```
generator = Generator() \
        .add_uniform() \
        .add_function(lambda h: h.e(1) ** 2, shadow=True) \
        .add_uniform() \
        .add_function(lambda h: h.e(2) + h.e(3))
```

La tabla 3.3 ilustra el conjunto resultante, donde se puede observar como la variable E_2 no está presente, pero sí el cómputo de su valor dentro de la variable E_4, es decir:

$$E_4 := E_2 + E_3 := E_1^2 + E_3$$

3.2.7 Generar conjunto de datos

Tras configurar un modelo causal con la ayuda de la clase `Generator`, conviene poder generar dicho conjunto. En Python es bastante habitual trabajar con la librería pandas y con su clase *DataFrame*, la cual es una estructura de datos a dos dimensiones que conviene perfectamente para datasets. La clase generador dispone entonces de un método `generate` para generar un DataFrame a partir de la definición de sus eventos, usable como a continuación:

```
df = generator.generate(samples=10000)
```

3.2.8 Extracción del grafo estructural

Con el fin de poder valorar algoritmos de exploración causal es indispensable conocer las relaciones que componen el conjunto de datos explorado. Describir dichas relaciones manualmente es posible pero no práctico. El generador es capaz de extraer las relaciones de forma automática a partir de su configuración de eventos. Para ello la clase `RelationFactory` explora uno a uno los eventos añadidos de tipo `Function`, extrae el código en formato texto gracias a la librería inspect de Python, busca referencias a otras variables a partir de los nombres de métodos `get_event` o `get_datetime` por ejemplo, y devuelve así un objeto `Relation` por cada relación encontrada, obteniendo finalmente una lista de relaciones. Adicionalmente contempla la dirección de las relaciones así como el desfase temporal si lo tiene.

3.2.9 Visualización del grafo estructural

Aunque no es necesario dentro de un proceso automático de integración continua, sí que resulta práctico poder visualizar las relaciones del modelo. Aprovechando la extracción automática de las relaciones del modelo podemos imprimir en pantalla el grado. La clase `RelationPlot` traduce las relaciones extraídas del generador en un grafo de la librería de Python *networkx* para dibujarlas en pantalla con matplotlib. Podemos entonces desde un objeto `Generator` visualizar su grafo correspondiente simplemente llamando al método `generator.plot_relations()` ilustrado en la imagen 3.4.

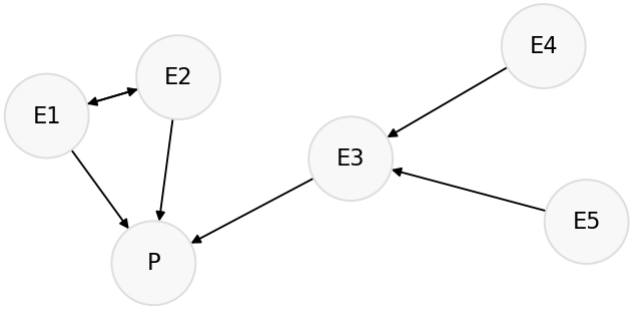

FIGURA 3.4: Visualización de un grafo estructural con RelationPlot.

Visualización de variables latentes

El método `generator.plot_relations()` representa por defecto los variables latentes marcandolos con un fondo rojo. Es posible no representar estas variables gracias al parámetro `include_shadow=False`, y el sistema es capaz de reducir correctamente las relaciones a las variables restantes. Como por ejemplo en la Figura 3.5, donde las variables E_1 y E_2 son padres de la variable Latent, pero estas se convierten en padres de la variable Effect tras esconder las variables latentes.

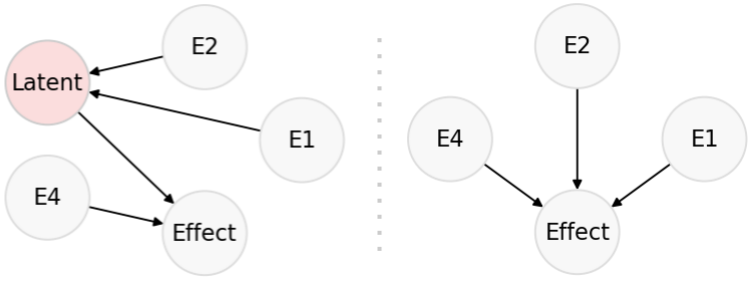

FIGURA 3.5: Visualización con variables latentes (izquierda) o sin (derecha).

3.3 Valoración de algoritmos

3.3.1 Descripción

El principal objetivo de este proyecto es poder medir el buen funcionamiento de uno o varios algoritmos de exploración causal de forma automática, como se haría con pruebas de código. Para ello es necesario poder configurar qué algoritmos y conjuntos serán valorados, y adicionalmente disponer de un script que haga la valoración. La clase responsable de dicha valoración es la clase `EvaluateLearner`, la cual ejecuta una exploración causal por cada algoritmo y cada conjunto configurado y mide en cada caso cuántas relaciones causales son encontradas correctamente. Adicionalmente se define un umbral mínimo de relaciones encontradas, de forma que la salida de la ejecución sea 0 (éxito) o 1 (fallido) [10] según todos los algoritmos superen o no el umbral. Se ilustra con el código a continuación:

```
exit(EvaluateLearner.run(scripts, datasets, threshold=90))
```

3.3.2 Visualización en tabla

La forma más compacta de presentar los resultados es presentarlos en formato de tabla. Por ello por defecto la ejecución de las exploraciones imprime una tabla de resumen con los resultados correspondientes. Asimismo, el *makefile* contiene el comando `run` que ejecuta las exploraciones con la clase `EvaluateLearner` e imprime en pantalla la tabla resumen como se puede apreciar en la Figura 3.6.

```
make run
```

dataset	samples	library	algorithm	found	erroneous	inverted	missing	time
Multiple Causes	1000	Pgmpy	PC	100%	0	0	0	32ms
Multiple Causes	1000	Pgmpy	GES	0%	0	2	0	213ms
Multiple Causes	1000	pyAgrum	GES	50%	0	1	0	12ms
Multiple Causes	1000	pyAgrum	TS	50%	0	1	0	9ms
Direct Causality	1000	Pgmpy	PC	100%	0	0	0	48ms
Direct Causality	1000	Pgmpy	GES	100%	0	0	0	273ms
Direct Causality	1000	pyAgrum	GES	100%	0	0	0	20ms
Direct Causality	1000	pyAgrum	TS	100%	0	0	0	26ms

FIGURA 3.6: Resultados de evaluación por EvaluateLearner en formato tabla.

El reporte detalla por cada combinación de conjunto y algoritmo el número de relaciones encontradas, cuántos errores ha cometido, así como el tiempo que ha tardado en ejecutarse.

3.3.3 Visualización en grafo

Otra manera más visual de analizar los resultados, es comparando el grafo original con el grafo aprendido por el algoritmo. Para ello se puede parametrizar clase `EvaluateLearner` en formato

de salida "grafo", de forma que imprima las estructuras causales en lugar de una tabla resumen. Se puede entonces visualizar en un *jupyter notebook* por ejemplo los diferentes grafos de las diferentes exploraciones, como ilustrado en la Figura 3.7 tras ejecutar el siguiente código:

```
EvaluateLearner.run(scripts, datasets, output='graph')
```

```
Multiple Effects - Discrete → GES (pyAgrum): 19%
```

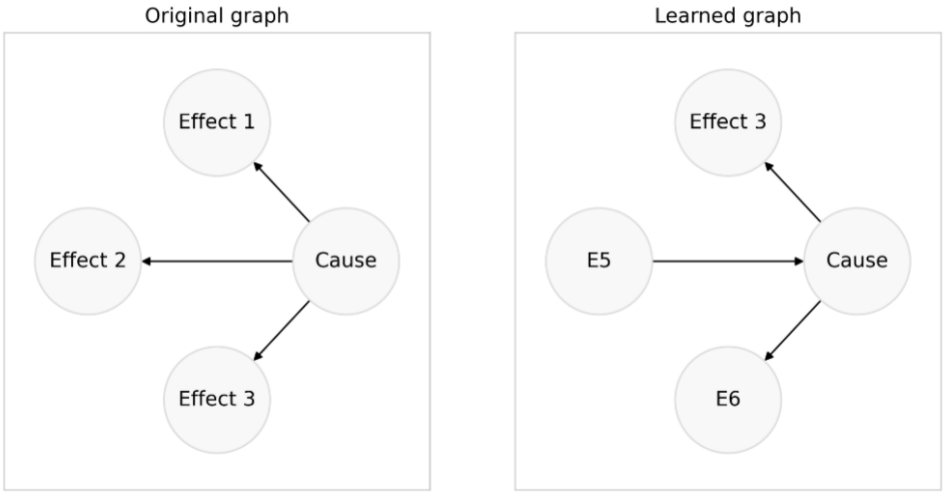

FIGURA 3.7: Resultados de evaluación por EvaluateLearner en formato grafo.

Resultados

4.1 Librerías y algoritmos

Para poder ejecutar exploraciones causales sobre conjuntos de datos relacionales, el framework desarrollado en este trabajo permite encapsular y añadir cualquier librería que permita aprender modelos estructurales. Adicionalmente, para disponer por defecto de algoritmos de exploración causal y experimentar con ellos, se integraron dos librerías de código abierto: pgmpy y pyAgrum.

4.1.1 Pgmpy

Esta librería se enfoca principalmente en representar modelos gráficos probabilísticos, determinando las dependencias entre variables aleatorias [1]. Dado un conjunto de datos y un modelo de red bayesiano, *pgmpy* puede parametrizar este último en función de los datos y realizar pruebas de inferencia. Pero el modelo en sí ha de ser definido manualmente. No obstante dispone de tres métodos de aprendizaje estructural, el primero basado en restricciones con una implementación del algoritmo *PC*, el segundo basado en puntuación con una implementación del algoritmo *GES* y finalmente una combinación de ambos métodos con una implementación del algoritmo *MMHC* [32]. Estos tres métodos están implementados en el framework y son usables tal como:

```
scripts = [
        PgmpyScript.PC(),
        PgmpyScript.GES(),
        PgmpyScript.MMHC(),
]
```

4.1.2 PyAgrum

La librería *pyAgrum* proporciona una interfaz en Python de la librería C++ *aGrUM*, la cual permite crear, gestionar y realizar operaciones de forma eficiente sobre redes bayesianas [33]. Igual que con pgmpy, esta librería es capaz de aprender los parámetros que describen un modelo bayesiano conocido, pero también implementa dos métodos de aprendizaje de estructura causal. Dispone de una implementación del algoritmo *GES* como método basado en puntuación, así como un método basado en restricción con una búsqueda local con lista tabú (TS) [4]. Ambos implemetadados y disponibles de la siguiente manera:

```
scripts = [
        PyAgrumScript.GES(),
```

```
        PyAgrumScript.TS(),
]
```

4.2 Conjuntos de datos unitarios

En el apartado 3.1 se describen los diferentes casos de relaciones causales unitarias. Con el ámbito de disponer de un abanico completo de conjuntos, se ha creado para cada tipo de relación su generador correspondiente, así como variantes con diferentes funciones de causa y efecto. Se detalla a continuación algunos ejemplos de la generación y exploración de estos conjuntos unitarios.

4.2.1 Causalidad directa

La relación de causalidad directa entre dos variables puede ser descrita por una función cualquiera tal que $Y := f(X) + N$. Este caso está modelado en la clase `DirectCausalityDataset`, y se describe a continuación un modelo con relación directa lineal, aplicando una función afín del tipo $y = ax + b + N$ entre las variables de causa y efecto.

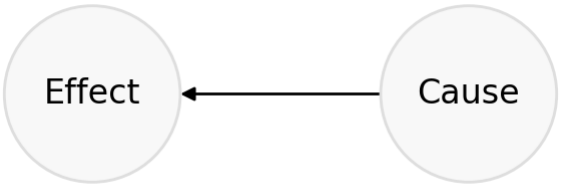

FIGURA 4.1: Grafo de modelo de causalidad directa con RelationPlot.

```
function = lambda h: 10 * h.e(1) + 5 + gauss(0, 5)

generator = Generator() \
        .add_discrete(10, label='Cause') \
        .add_discrete(10) \
        .add_discrete(10) \
        .add_discrete(10) \
        .add_function(function, label='Effect', round=0)
```

En el bloque de código anterior, se construye un objeto `Generator` a partir de la función `function`, la cual vincula las variables en posición 1 y 5 de tal forma que $V_5 := 10V_1 + 5 + N$, donde N es un ruido gausiano de media 0 y de desviación típica 5.
Adicionalmente se nombran *Cause* la variable 1, y *Effect* la variable 5, y el resultado es redondeado con 0 decimales para obtener una valor discreto. Todas las variables salvo el efecto son variables discretas aleatorias entre 0 y 9.

TABLA 4.1: Conjunto de datos con causalidad directa linear.

Cause	E2	E3	E4	Effect
4	6	7	5	39
4	3	3	9	44
6	4	6	4	67
3	0	0	6	40
0	9	9	6	10

La Tabla 4.1 ilustra algunas muestras generadas con esta configuración, así como la Figura 4.1 representa el grafo estructural correspondiente tras ejecutar el código a continuación:

```
generator.plot_relations()
```

Finalmente, se ejecuta una exploración de cada algoritmo sobre este conjunto con 1000 muestras, y tal como se aprecia en la Figura 4.2, todos los algoritmos encontraron correctamente la relación y la orientación entre la primera y última variable.

library	algorithm	found	erroneous	inverted	missing
Pgmpy	PC	100%	0	0	0
Pgmpy	GES	100%	0	0	0
pyAgrum	GES	100%	0	0	0
pyAgrum	TS	100%	0	0	0

FIGURA 4.2: Resultado de exploración de un conjunto de causalidad directa linear.

Es interesante destacar que si se ejecuta las veces suficientes la exploración sobre las mismas muestras, es decir usando siempre los mismos valores ya generados, en alguna ocasión el algoritmo *PC* de *pgmpy* invierte la relación como se puede ver el la Figura 4.3.

library	algorithm	found	erroneous	inverted	missing
Pgmpy	PC	0%	0	1	0
Pgmpy	GES	100%	0	0	0
pyAgrum	GES	100%	0	0	0
pyAgrum	TS	100%	0	0	0

FIGURA 4.3: Resultado de exploración de un conjunto de causalidad directa linear.

El framework permite igualmente visualizar el resultado de la exploración de forma gráfica como ilustrado en las Figuras 4.4 y 4.5, donde se puede observar la relación invertida del algoritmo *PC* y la correcta del algoritmo *GES* de la librería *pyAgrum*.

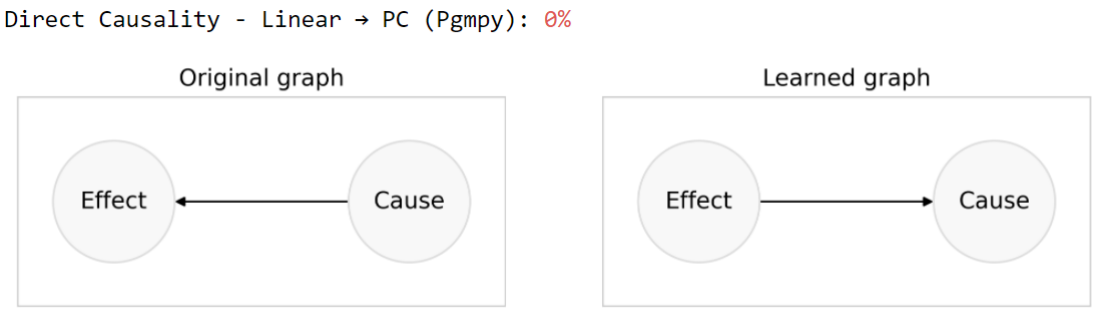

FIGURA 4.4: Algoritmo PC (Pgmpy) con causalidad directa linear.

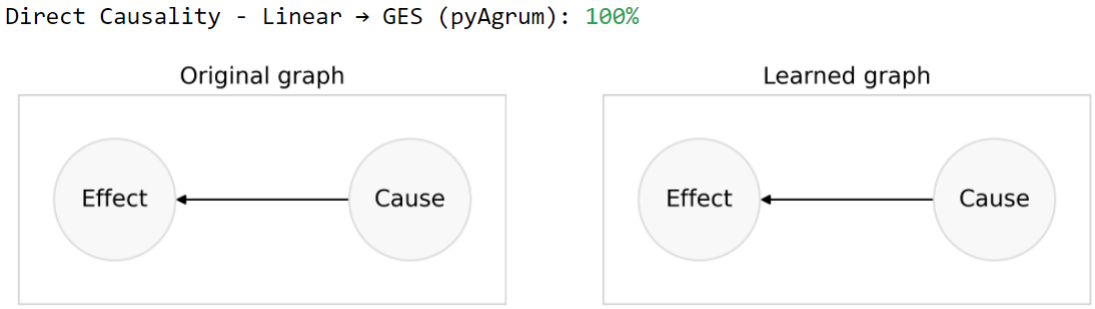

FIGURA 4.5: Algoritmo GES (pyAgrum) con causalidad directa linear.

4.2.2 Causalidad múltipe

La clase `MultipleCausesCausalityDataset` encapsula las configuraciones de un conjunto de datos con múltiples causas y un único efecto. En este caso en lugar de un ejemplo basado en una función continua, se detalla el uso de condiciones discretas o probabilísticas, donde el valor de las causas determina la probabilidad del efecto de ser 0 o 1.
Se define entonces la variable efecto como binaria, y la probabilidad de que valga 0 o 1 es función de la media del valor de dos otras variables. Para ello el valor de ambas variables de causa es reducido a un intervalo simétrico centrado en cero sobre el cual se aplica una función sigmoide de forma a potenciar los extremos.

La Figura 4.6 muestra la evolución de la probabilidad de valer 1 según el valor de una variable aleatoria discreta entre 0 y 9.

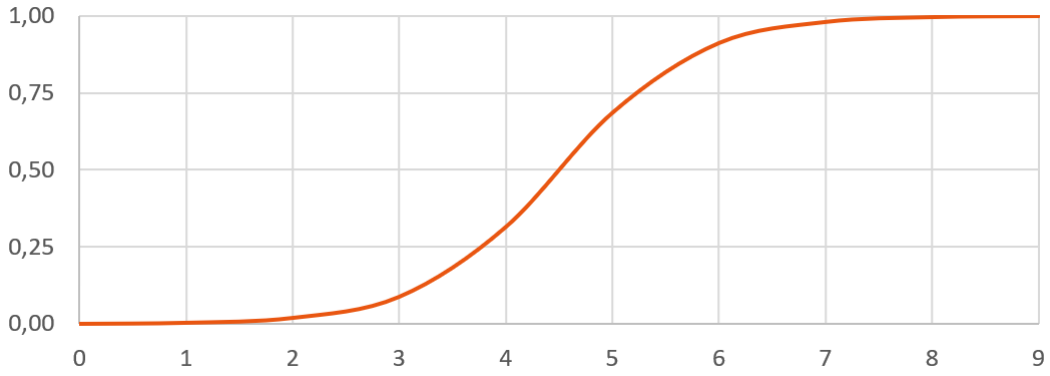

FIGURA 4.6: Distribución de probabilidad según el valor de causa.

Para entenderlo mejor, el código a continuación corresponde a la función de causa y efecto entre las variables. La propiedad `sharpness` corresponde al afilado de la sigmoide.

```
def function(h: History):
    sharpness = 7

    # Transform range from [0, 9] to [-sharpness, sharpness].
    value1 = (h.e(1) - 0) / (9 - 0) * (sharpness * 2) - sharpness
    # Apply a sigmoid probability distribution.
    probability1 = 1 / (1 + math.exp(-value1))

    # Transform range from [0, 19] to [-sharpness, sharpness].
    value2 = (h.e(2) - 0) / (19 - 0) * (sharpness * 2) - sharpness
    # Apply a sigmoid probability distribution.
    probability2 = 1 / (1 + math.exp(-value2))

    probability = 1 / 2 * (probability1 + probability2)
    weights = (1 - probability, probability)

    return Discrete(weights=weights).generate()

generator = Generator() \
    .add_discrete(10, label='Cause 1') \
    .add_discrete(20, label='Cause 2') \
    .add_discrete(5) \
    .add_discrete(3) \
    .add_function(function, label='Effect')
```

Resulta interesante destacar que con un conjunto de 100 muestras el algoritmo PC de pgmy es capaz de detectar correctamente todas las relaciones de forma consistente, como ilustrado en la Figura 4.7 mientras que ninguno de los otros algoritmos lo logran, como por ejemplo el algoritmo GES de pgmpy también, ilustrado en la Figura 4.8.

```
Multiple Causes - Discrete → PC (Pgmpy): 100%
```

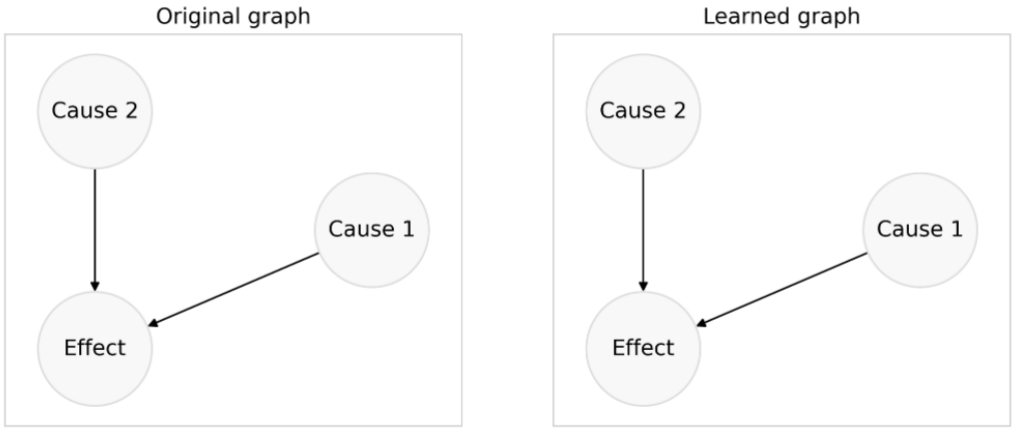

FIGURA 4.7: Algoritmo PC (pgmpy) con conjunto de causas múltiples.

```
Multiple Causes - Discrete → GES (Pgmpy): 16%
```

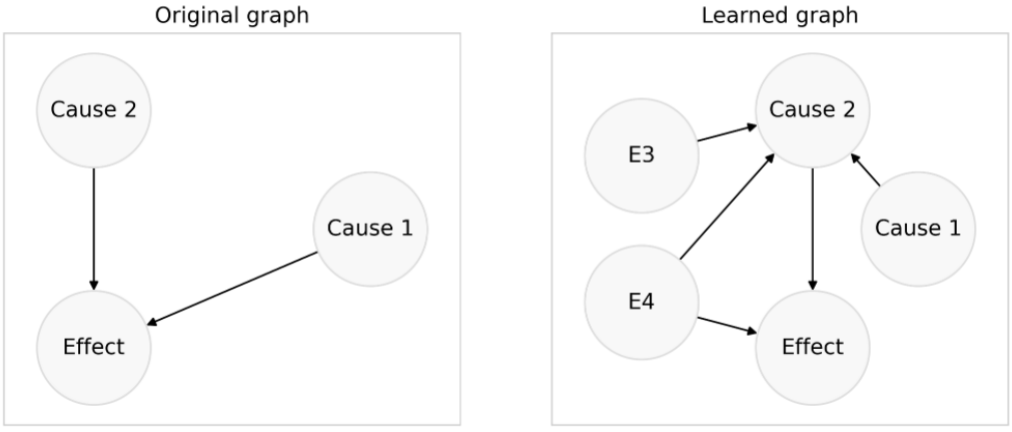

FIGURA 4.8: Algoritmo GES (pgmpy) con conjunto de causas múltiples.

A partir de 1000 muestras todos los algortimos detectan las relaciones correctas, pero a excepción del algorimo PC no logran ser consitentes sobre la dirección de las relaciones incluso con 50.000 muestras como ilustrado en la Figura 4.8.o la tabla resultante en la Figura 4.10.

```
Multiple Causes - Discrete → GES (pyAgrum): 50%
```

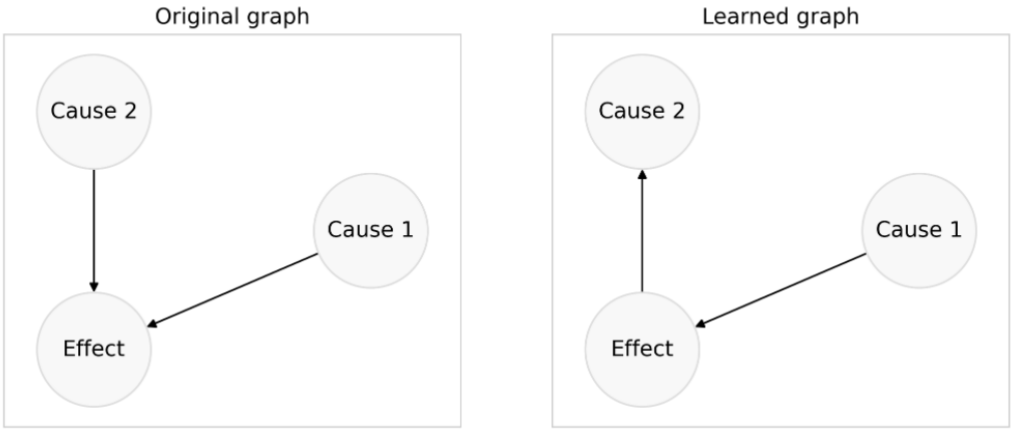

FIGURA 4.9: Algoritmo GES (pyAgrum) sobre un conjunto de causas múltiples de 50.000 muestras.

library	algorithm	found	erroneous	inverted	missing
Pgmpy	PC	100%	0	0	0
Pgmpy	GES	0%	0	2	0
pyAgrum	GES	50%	0	1	0
pyAgrum	TS	50%	0	1	0

FIGURA 4.10: Resultado de exploraciones en formato tabla sobre un conjunto de causas múltiples de 50.000 muestras.

4.2.3 Causalidad circular

Un caso especial es el caso de causalidad circular, donde impone el uso de un desfase temporal. El conjunto descrito a continuación relaciona 4 eventos en cadena donde cada evento corresponde al valor del evento anterior ± 1 de la muestra anterior, de tal forma que el evento en posición 4 es causa del evento en posición 1.

```
functions = [
        lambda h: (h.e(4, delay=1) or 0) + 0.5 + gauss(0, 1),
        lambda h: (h.e(1, delay=1) or 0) + 0.5 + gauss(0, 1),
        lambda h: (h.e(2, delay=1) or 0) + 0.5 + gauss(0, 1),
        lambda h: (h.e(3, delay=1) or 0) + 0.5 + gauss(0, 1),
]

generator = Generator() \
        .add_function(functions[0], label='1') \
        .add_function(functions[1], label='2') \
```

```
        .add_function(functions[2], label='3') \
        .add_function(functions[3], label='4') \
        .add_discrete(10) \
        .add_discrete(10)
```

La Figura 4.11 ilustra el grafo del modelo, asi como un exploración con el algoritmo PC de pgmpy. No detecta el bucle entero pero si que existe una relación entre las variables 1,2 y 4.

```
Circular Causality - Discrete → PC (Pgmpy): 25%
```

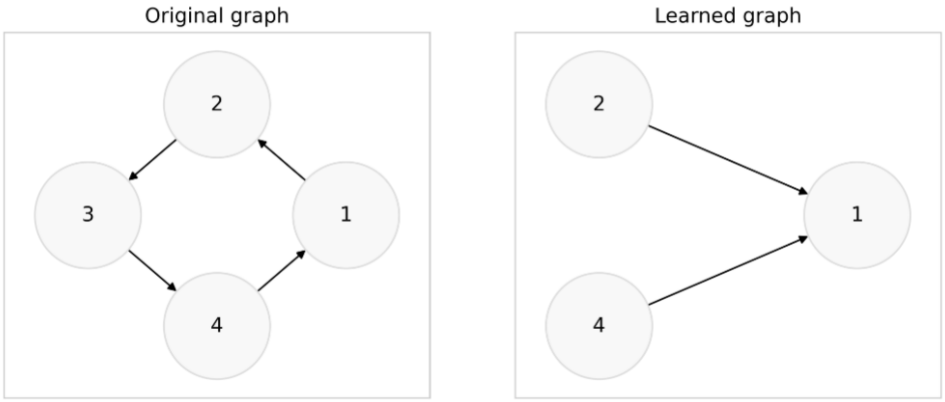

FIGURA 4.11: Resultado de la exploración del algoritmo PC (pgmpy) sobre un conjunto de causalidad circular.

Conclusiones y desarrollos futuros

5.1 Algoritmos de Causal Discovery

Este trabajo integra por defecto dos librerías de exploración causal: pgmpy y pyagrum. Pero realmente se han probado múltiples librerías y algoritmos como *dowhy*, *CausalImpact*, *CausalInference* o *medil* que no han sido incluidas de base en el framework por no permitir exploraciones sin definir previamente un modelo de grafo o definir específicamente una variable de efecto. Adicionalmente en algunos casos son necesarios instalar más de 10Gb de dependencias resultando en una instalación básica del proyecto demasiada pesada, la cual se desea ser minimalista. Aún así, todos comparten ciertas limitaciones detalladas a continuación.

5.1.1 Especificidad

A día de hoy ningún algoritmo es generalista, cada uno resuelve correctamente un rango reducido de casos. Como se ha podido comprobar en las pruebas efectuadas, los resultados varían entre algoritmos según los tipos de relaciones del conjuntos de datos, e incluso entre librerías con implementaciones de una misma técnica, como es el caso del *GES* de *pgmpy* o *pyAgrum* que no dan los mismos resultados con los mismos conjuntos. Actualmente y de forma general, los algoritmos de causal discovery son muy específicos y por este mismo motivo existen tantas variantes.

5.1.2 Repetibilidad

Tanto la generación de las muestras de los conjuntos, como los propios algoritmos contienen elementos aleatorios, y por tanto dos ejecuciones consecutivas no siempre dan el mismo resultado. Se observa principalmente en la selección de la orientación de las relaciones, lo cual es aceptable al ser teóricamente imposible diferenciar la causa y el efecto en relaciones directas [19] (aunque existen técnicas para hacerlo [17]), también varían en algunos casos la estructura causal encontrada sin tener en cuenta la orientación de las relaciones. Aunque esta última variabilidad se reduce con el aumento de la cantidad de muestras, se puede observar casos donde la cantidad de muestras no mejora la estabilidad del resultado.

5.1.3 Tiempo

Un punto importante a tener en cuenta dentro de un proceso de Integración continua es el tiempo de ejecución de las pruebas. Justamente destaca aquí que el tiempo de ejecución se dispara considerablemente no solo con la cantidad de variables añadidas, sino sorprendentemente

también con el tipo de relación. Dos configuraciones de conjunto idénticas, a excepción de la función de causa y efecto, con el mismo número de variables y muestras, pueden necesitar un tiempo de exploración muy dispares. Pero el impacto principal es el número de variables contenidas por el conjunto, más que la cantidad de muestras, el cual puede rápidamente cambiar de milisegundos a minutos o decenas de minutos.

5.1.4 Recursos

Otro elemento crítico además del tiempo, es la infraestructura o recursos necesarios para ejecutar las pruebas. Dependiendo del tipo de conjunto, de la cantidad de muestras o del número de variables, e incluso de una exploración puntual, los algoritmos pueden necesitar reservar más memoria de la disponible en la máquina, haciendo directamente fallar la ejecución y sin finalizar. Con las pruebas realizadas los algoritmos llegaron a necesitar hasta 12Gb de memoria RAM. Para evitarlo las propias librerías de causal discovery permiten parametrizar las exploraciones de forma a limitar la profundidad de estas, con la contrapartida de reducir su precisión.

5.2 Limitaciones del framework

El framework resultante de este trabajo es funcional y se podría imaginar integrar en un proceso de pruebas automáticas sin demasiado esfuerzo, teniendo en cuenta los tiempos y recursos necesarios para las exploraciones. Tras el análisis y evaluación de las diferentes técnicas y librerías de exploración causal, este proyecto resulta de interés como base de seguimiento y comprobación para el desarrollo de un algoritmo de aprendizaje de estructura causal generalista. Principalmente en el sentido que permite crear y valorar de forma unificada y centralizada todos los tipos de relaciones causales. Para ello se pueden plantear ciertas mejoras descritas a continuación.

5.2.1 Identificador de evento

La clase `Generator` permite vincular cualquier variable con cualquiera para crear las relaciones necesarias del conjunto, y para ello los eventos se referencian por su posición. No resulta ser un problema hasta que sea necesario modificar el orden en el cual se construyen los eventos, en cual caso se ha de actualizar el índice de dichas referencias. Sería conveniente poder definir por evento un "identificador" textual usable para referenciarlos. Ahora mismo se puede asignar un *nombre* o *etiqueta* a los eventos con el parámetro `delay`, pero este el nombre se ha de considerar mutable y no tendría que ser usado como identificador.

5.2.2 Media de ejecuciones

Cómo abordado en el punto 5.1.2, cada exploración de un mismo conjunto con un mismo algoritmo varía de forma aleatoria, y asimismo sus resultados. Con el objetivo de maximizar la estabilidad del resultado positivo o negativo de cada ejecución, se podría definir una cantidad de repeticiones por cada algoritmo y conjunto, de forma a usar la media de los resultados correspondientes.

5.2.3 Visualización de los desfases temporales

En su estado actual, la clase `RelationBuilder` detecta los desfases temporales a la vez que extrae las relaciones desde las configuraciones de un *generador*. Pero a pesar de disponer del valor, no tiene en cuenta estos desfases a la hora de imprimir el grafo estructural con `RelationPlot`, y permitir así visualizarlos. Los algoritmos de exploración de las librerías integradas con el proyecto no devuelven información sobre desfase temporal, y por tanto no es posible comparar y valorar dicho desfase. No obstante, sería interesante poder visualizarlos sobre el grafo original, aunque implique tener que modificar la generación del grafo, y complicar posiblemente la lectura.

5.2.4 Casuística de detección de relación

El framework es capaz de extraer automáticamente y correctamente las relaciones entre eventos a partir de las configuraciones del *generador*. Mientras esta extracción resulta práctica y funcional tiene la limitación de sólo poder encontrar relaciones desde el uso de los métodos definidos en la clase `History`. En el caso en que se acceda directamente a los valores del histórico, sin usar los métodos de la clase, como por ejemplo `history.events[1][0]` en lugar de `history.get_event(0, delay=1)`, entonces no será detectada la relación. Es imaginable extraer la relación de esta forma igualmente, pero añade una complejidad adicional no necesaria actualmente en este trabajo.

Dockerfile

```
FROM debian:buster-slim

ARG DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt install -qy \
    wget

# Install R.
RUN apt install -qy r-base

# Install SSH.
RUN apt install -qy openssh-server

# Install Python through miniconda.
RUN wget -qO miniconda.sh https://repo.anaconda.com/miniconda/...-x86_64.sh && \
    /bin/bash miniconda.sh -b -p /usr/lib/miniconda3 && \
    rm -f miniconda.sh && \
    ln -s /usr/lib/miniconda3/bin/conda /usr/bin/conda && \
    conda update -n base -c defaults conda

# PyAgrum dependencies.
RUN apt install -qy graphviz

# Clean up.
RUN apt-get remove -y \
    wget \
    && apt-get clean

# Keep container up.
CMD ["tail", "-f", "/dev/null"]
```

Makefile

```
# Container SSH port
SSH_PORT=22000
# Container Jupyter port
JUPYTER_PORT=22001
# Image and container tags (Causal Discovery Unit Testing)
IMAGE=cdut.img
CONTAINER=cdut.con
# Python virtual environment relative path
ENV_PATH=.env
# Path to the evaluation script
SCRIPT_PATH=src/ui/cli.py
# Execute container as root
EXEC=docker exec -it $(CONTAINER)
# Execute container as current user
EXEC_U=docker exec -itu $$(id -u) $(CONTAINER)


stop:
    # Remove container instead of stopping it to free up its tag.
    # Always true to ignore error when container does not exist.
    docker rm -f $(CONTAINER) || true

build:
    docker build -t $(IMAGE) -f dockerfile .

create:
    $(EXEC_U) conda env create -f environment.yml --force --prefix $(ENV_PATH)

start: stop build
    # Run container in the background.
    docker run -v $(PWD):/app -w /app -p $(SSH_PORT):22 \
      -p $(JUPYTER_PORT):$(JUPYTER_PORT) --name $(CONTAINER) -di $(IMAGE)
    # Create user 'app' as current user (same id).
    $(EXEC) useradd -s /bin/bash -u $$(id -u) -Md /app app
    $(EXEC) /bin/bash -c "echo app:app | chpasswd"
```

```
    # Start ssh service here, as it cannot be started from the docker image.
    $(EXEC) service ssh start
    # Remove previous SSH key.
    ssh-keygen -f "$(HOME)/.ssh/known_hosts" -R "[localhost]:$(SSH_PORT)"
    # Create conda local environment only once.
    if [ ! -d "$(ENV_PATH)" ]; then make create; fi
    # Make environment Python available in PATH.
    $(EXEC) ln -fs /app/$(ENV_PATH)/bin/python /usr/bin/

root:
    $(EXEC) /bin/bash -l

ssh:
    # Asks for password: app. Avoid having ssh key issues.
    ssh -o PubkeyAuthentication=no app@localhost -p $(SSH_PORT)

conda:
    $(EXEC_U) conda $(filter-out $@,$(MAKECMDGOALS)) --prefix $(ENV_PATH)

pip:
    $(EXEC_U) $(ENV_PATH)/bin/pip $(filter-out $@,$(MAKECMDGOALS))

test:
    $(EXEC_U) bash -c "PYTHONPATH=src python -m unittest"

jupyter:
    # Use the link by IP, not by container name.
    $(EXEC_U) bash -c "PYTHONPATH=src $(ENV_PATH)/bin/jupyter \
      $(filter-out $@,$(MAKECMDGOALS)) --ip=0.0.0.0 --port=$(JUPYTER_PORT) \
      --no-browser"

run:
    $(EXEC_U) bash -c "PYTHONPATH=src python $(SCRIPT_PATH)"
```

Bibliografía

[1] Ankur Ankan and Abinash Panda. "pgmpy: Probabilistic graphical models using python". In: *Proceedings of the 14th Python in Science Conference (SCIPY 2015). Citeseer*. Vol. 10. Citeseer. 2015. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.986.409&rep=rep1&type=pdf.

[2] Ned Batchelder. *Coverage.py*. 2009. URL: https://coverage.readthedocs.io/en/coverage-5.3/.

[3] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[4] Stefano Beretta et al. "Learning the structure of Bayesian Networks: A quantitative assessment of the effect of different algorithmic schemes". In: *Complexity* 2018 (2018). URL: https://arxiv.org/pdf/1704.08676.pdf.

[5] David Maxwell Chickering. "Optimal structure identification with greedy search". In: *Journal of machine learning research* 3.Nov (2002), pp. 507–554. URL: https://www.jmlr.org/papers/volume3/chickering02b/chickering02b.pdf.

[6] John Concato, Nirav Shah, and Ralph I Horwitz. "Randomized, controlled trials, observational studies, and the hierarchy of research designs". In: *New England journal of medicine* 342.25 (2000), pp. 1887–1892. URL: https://www.nejm.org/doi/full/10.1056/nejm200006223422507.

[7] Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[8] Martin Fowler and Matthew Foemmel. *Continuous integration*. 2006. URL: https://moodle2019-20.ua.es/moodle/pluginfile.php/2228/mod_resource/content/2/martin-fowler-continuous-integration.pdf.

[9] Clark Glymour, Kun Zhang, and Peter Spirtes. "Review of causal discovery methods based on graphical models". In: *Frontiers in genetics* 10 (2019), p. 524. URL: https://www.frontiersin.org/articles/10.3389/fgene.2019.00524/full.

[10] GNU. *Manual - Exit status*. Acceso: 2020-10-05. 1993. URL: https://www.gnu.org/software/libc/manual/html_node/Exit-Status.html.

[11] Isabelle Guyon et al. "Causality workbench". In: *Causality in the sciences*. Oxford University Press, 2011. URL: https://experts.umn.edu/en/publications/causality-workbench.

[12] Patrik O Hoyer et al. "Estimation of causal effects using linear non-Gaussian causal models with hidden variables". In: *International Journal of Approximate Reasoning* 49.2 (2008), pp. 362–378. URL: https://www.sciencedirect.com/science/article/pii/S0888613X08000212.

[13] Kosuke Imai, Dustin Tingley, and Teppei Yamamoto. "Experimental designs for identifying causal mechanisms". In: *Journal of the Royal Statistical Society: Series A (Statistics in Society)* 176.1 (2013), pp. 5–51. URL: https://imai.fas.harvard.edu/research/files/Design.pdf.

[14] J. Runge J. Muñoz-Marí G. Mateo and G. Camps-Valls. "CauseMe". In: *An online system for benchmarking causal discovery methods*. 2020. URL: https://causeme.uv.es/.

[15] Mike Jackson. *Software Carpentry: Automation and Make*. 2016. URL: https://swcarpentry.github.io/make-novice.

[16] Jiuyong Li, Lin Liu, and Thuc Duy Le. *Practical approaches to causal relationship exploration*. Springer, 2015.

[17] Joris M Mooij et al. "Distinguishing cause from effect using observational data: methods and benchmarks". In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1103–1204. URL: https://jmlr.org/papers/volume17/14-518/14-518.pdf.

[18] Mohammed Ombadi et al. "Evaluation of methods for causal discovery in hydrometeorological systems". In: *Water Resources Research* 56.7 (2020), e2020WR027251. URL: https://agupubs.onlinelibrary.wiley.com/doi/abs/10.1029/2020WR027251.

[19] Jonas Peters, Dominik Janzing, and Bernhard Schölkopf. *Elements of Causal Inference : Foundations and Learning Algorithms*. The MIT Press, 2017. ISBN: 9780262037310. URL: https://library.oapen.org/bitstream/id/056a11be-ce3a-44b9-8987-a6c68fce8d9b/11283.pdf.

[20] President and Fellows of Harvard College. *Thinking About Relational Causality*. 2003. URL: http://causalpatterns.org/resources/airpressure/pdfs/s3_resources_thinking.pdf.

[21] Tenko Raykov and George A Marcoulides. *A first course in structural equation modeling*. Routledge, 2012. URL: https://books.google.es/books?id=K-pkAgAAQBAJ&printsec=frontcover&hl=es&source=gbs_ge_summary_r&cad=0#v=onepage&q&f=false.

[22] Fritz K Ringer. "Causal analysis in historical reasoning". In: *History and theory* 28.2 (1989), pp. 154–172. URL: https://www.jstor.org/stable/2505033.

[23] Shohei Shimizu et al. "A linear non-Gaussian acyclic model for causal discovery". In: *Journal of Machine Learning Research* 7.Oct (2006), pp. 2003–2030. URL: http://www.jmlr.org/papers/v7/shimizu06a.html.

[24] Julian L Simon. "The concept of causality in economics". In: *Kyklos* 23.2 (1970), pp. 226–254. URL: https://econpapers.repec.org/article/blakyklos/v_3a23_3ay_3a1970_3ai_3a2_3ap_3a226-254.htm.

[25] Karamjit Singh et al. "Comparative benchmarking of causal discovery techniques". In: *arXiv preprint arXiv:1708.06246* (2017). URL: https://arxiv.org/pdf/1708.06246.pdf.

[26] Peter Spirtes et al. *Causation, prediction, and search*. MIT press, 2000.

[27] Diana Suhr. "The basics of structural equation modeling". In: *Presented: Irvine, CA, SAS User Group of the Western Region of the United States (WUSS)* (2006). URL: https://scholar.google.es/scholar?hl=es&as_sdt=0%2C5&q=The+Basics+of+Structural+Equation+Modeling+Diana+Suhr%2C+Ph.D.+University+of+Northern+Colorado+&btnG=.

[28] David Thomas and Andrew Hunt. *The pragmatic programmer*. Addison-Wesley, 2000. URL: https://file1.largepdf.com/file/2020/04/12/The_Pragmatic_Programmer_20th_Anniversary_-_Andrew_Hunt.pdf.

[29] David Perkins Tina Grotzer. *Causal Patterns*. Acceso: 2020-09-15. 2008. URL: https://www.cfa.harvard.edu/smg/Website/UCP/pdfs/SixCausalPatterns.pdf.

[30] David Perkins Tina Grotzer. *Causal Patterns in Science - Ecosystems Curriculum*. Acceso: 2020-09-15. 2010. URL: http://causalpatterns.org/resources/ecosystems/resources_curr_challenges.php.

[31] Sofia Triantafillou and Ioannis Tsamardinos. "Score-based vs Constraint-based Causal Learning in the Presence of Confounders." In: *CFA@ UAI*. 2016, pp. 59–67. URL: http://www.its.caltech.edu/~fehardt/UAI2016WS/papers/Triantafillou.pdf.

[32] Ioannis Tsamardinos, Laura E Brown, and Constantin F Aliferis. "The max-min hill-climbing Bayesian network structure learning algorithm". In: *Machine learning* 65.1 (2006), pp. 31–78. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.4729&rep=rep1&type=pdf.

[33] Pierre-Henri Wuillemin. *pyAgrum Documentation*. Acceso: 2020-10-31. 2020. URL: https://readthedocs.org/projects/pyagrum/downloads/pdf/latest/.

[34] Eric P. Xing. *Lecture 3 : Representation of Undirected Graphical Model*. University Lecture. 2017. URL: https://www.cs.cmu.edu/~epxing/Class/10708-17/notes-17/10708-scribe-lecture3.pdf.