Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Optimización de ASV para árboles de decisión

Tesis de Licenciatura en Ciencias de la Computación

Ezequiel Companeetz

Director: Santiago Cifuentes
Codirector: Sergio Abriola
Buenos Aires, 2025

Optimización de ASV para árboles de decisión

En esta tesis se aborda el problema de la explicabilidad en modelos de aprendizaje automático mediante métodos de feature attribution.

En particular, se estudia una variante de los Shapley values conocida como Asymmetric Shapley Values (ASV), que permite incorporar conocimiento causal en la explicación de modelos de forma model-agnostic. A partir del análisis de su complejidad, se demuestra que el cálculo exacto de ASV es polinomial en modelos cuya distribución de entrada está representada por una red bayesiana del tipo Naive Bayes, en contraste con SHAP, que es #P-hard aún en este caso restringido.

Con el objetivo de extender estos resultados a clases más generales de redes bayesianas, se introduce una noción de clases de equivalencia sobre los órdenes topológicos del grafo causal subyacente, lo cual permite reducir drásticamente el número de permutaciones necesarias para computar ASV. Se presenta un algoritmo polinomial en el número de clases para identificarlas, y se implementa un esquema de cómputo exacto de ASV basado en estas clases. Además, se propone un nuevo método para computar en tiempo polinomial la predicción esperada de un árbol de decisión, sobre una distribución dada por una red bayesiana arbitraria, permitiendo así evaluar el algoritmo desarrollado para el cómputo de ASV en estos modelos.

Por último, se propone un algoritmo aproximado para calcular el ASV en familias de DAG's causales del tipo polytree. Para ello, se desarrolla un algoritmo de muestreo aleatorio de órdenes topológicos de polytrees. Estos resultados respaldan la viabilidad del enfoque propuesto en estructuras causales realistas, y se contrastan empíricamente con SHAP tanto en precisión como en eficiencia computacional.

Palabras clave: Explicabilidad, Árboles de decisión, Asymmetric Shapley Values (ASV), Shap, Polytrees, Ordenes topológicos

ASV optimization for Decision Trees

This thesis addresses the problem of explainability in machine learning models through feature attribution methods.

In particular, it studies a variant of the Shapley values known as Asymmetric Shapley Values (ASV), which allows for the incorporation of causal knowledge into model-agnostic explanations. Through a complexity analysis, it is shown that the exact computation of ASV is polynomial for models whose input distribution is represented by a Naive Bayes Bayesian network, in contrast to SHAP, which is #P-hard even in this restricted case.

To extend these results to more general classes of Bayesian networks, a notion of equivalence classes over the topological orderings of the underlying causal graph is introduced. This approach drastically reduces the number of permutations required to compute ASV. A polynomial-time algorithm is presented to identify these classes, and an exact ASV computation scheme based on them is implemented. Additionally, a new method is proposed to compute the expected prediction of a decision tree in polynomial time over a distribution given by an arbitrary Bayesian network, thereby enabling the evaluation of the developed ASV computation algorithm on these models.

Finally, an approximate algorithm is proposed to compute the ASV in families of causal DAGs of the polytree type. To this end, a random sampling algorithm for topological orderings of polytrees is developed. These results support the feasibility of the proposed approach in realistic causal structures and are empirically compared with SHAP in terms of both accuracy and computational efficiency.

Keywords: Explainability, Decision Trees, Asymmetric Shapley Values (ASV), Shap, Polytrees, Topological Orders

Agradecimientos

Primero que nada, agradezco a mis directores, Santiago y Sergio, los cuales no sólo me ayudaron a encontrar un tema acorde a lo que buscaba. Sino que también me dieron excelentes comentarios y correcciones que me guiaron en el proceso de realizar esta tesis.

Muchísimas gracias a mi familia, la cual me acompaño en todo este trayecto universitario y sin la cual no hubiera podido llegar adonde estoy. Por la contención, el aguante y por estar ahí.

Muchas gracias a los jurados, Pablo Riera y Eric Brandwein, por tomarse el tiempo de leer y evaluar esta tesis.

Gracias a Carlos Miguel Soto y Pablo Terlisky por la colaboración con el algoritmo de conteo de órdenes topológicos en polytrees.

Gracias a mis amigos y compañeros de cursada, los cuales me contagiaron su pasión: Rama, Male, Franquito, Lombi, Chara, Tomi, Pedrito, Fede, Pau, La Plebe, Amigos de Berto y todos los amigos de un cuatri que hice. Realmente no hubiera podido hacerlo sin su apoyo, desde guías resueltas a excelentes compañeros de trabajos prácticos.

Gracias también a Exactas por ser un espacio tan bello tanto para juntarse a estudiar, como para jugar al voley o dar clase.

También estoy muy agradecido de los excelentes docentes del departamento de computación con los que tuve el placer de cursar, los cuales daban clases que te desafiaban, despertaban tu curiosidad y lograban transmitirte su pasión. Por último, gracias a la universidad pública por permitir que esto suceda.

Índice

1. Introducción

Las capacidades de los distintos modelos de inteligencia artificial (IA) han ido creciendo exponencialmente en los últimos años. Tareas que creíamos imposibles de realizar a través de estas técnicas, como la resolución de problemas matemáticos complejos, hoy en día pueden ser resueltas por los LLMs [**?**]. No sólo aumentó la complejidad de las tareas que pueden resolver, sino también la complejidad de sus arquitecturas y de como están compuestos. Los modelos de hace unos años tenían 1e8 parámetros, mientras que hoy en día ya llegan a 1e12. [**?**]. El problema que esto genera es que cada vez es más difícil entender como funcionan y como obtienen las respuestas que nos dan. Esto es algo sumamente importante, ya que si uno está utilizando estas técnicas para diagnósticos médicos u otras situaciones igual de delicadas, es fundamental entender como los modelos alcanzan sus conclusiones . Ahí es donde entra en juego el área de XAI, Explainable Artificial Intelligence.

El objetivo principal de esta área de investigación consiste en encontrar una explicación para las decisiones o predicciones de los distintos modelos de IA, para poder entender el razonamiento detrás de las mismas. Dentro de esta área hay dos ramas principales, la explicabilidad y la interpretabilidad. La primera se centra en obtener estas explicaciones, y es el área en la cual nos centramos en este trabajo. La segunda en entender a los modelos y su representación interna [**?**]. Estas explicaciones no son útiles únicamente en tanto permiten que los usuarios entiendan el porqué de la respuesta, sino también debido a que ayudan a detectar errores o sesgos indeseados que se hayan generado durante el entrenamiento.

Dentro del área de explicabilidad hay varios tipos de métodos [**?**], los cuales pueden agruparse de acuerdo a distintos ejes:

- *Modelo*
  - Agnóstico: Son los métodos que se pueden aplicar a todo tipo de modelos. Por ejemplo en SHAP, el método que describiremos más adelante, su valor no depende de la implementación interna del modelo.
  - Específico: Son los métodos que tienen una implementación que está acoplada al modelo que buscan explicar, como en el caso de *Tree-Shap*, que en su implementación se utilizan los caminos del árbol para calcular su valor, aprovechando propiedades de la estructura del modelo.
- *Alcance*
  - Explicación global: Busca descubrir cualidades o comportamientos que el modelo tenga en todas sus predicciones. Por ejemplo, para entender por qué un modelo le otorga un préstamo a un individuo, se observa que el nivel de ingresos siempre es un feature relevante.
  - Explicación local: Busca descubrir cualidades o comportamiento que el modelo tenga para un conjunto reducido de instancias del dataset. Por ejemplo, para entender por qué un modelo le otorga un préstamo a un individuo, podemos encontrar que para las personas casadas, la cantidad de hijos es un feature que toma más relevancia.
- *Tipo de explicación:* Hay varias categorías del tipo de explicación que puede tener un método, por ejemplo *feature importance*, el cual asigna valores a cada feature. También hay *interpretaciones visuales* como saliency maps o correlations plots. Incluso hay métodos que generan un modelo más simple a partir del modelo original, como *Lime*.

  Cabe destacar que no existe un consenso absoluto sobre qué constituye una "explicación" en el contexto de la inteligencia artificial. Muchas de las técnicas mencionadas —como feature attribution o interpretaciones visuales— son aproximaciones pragmáticas a un problema mucho más amplio y complejo, vinculado a cuestiones filosóficas sobre comprensión, causalidad e interpretación humana [**?**, **?**]. En este sentido, lo que hoy se considera una explicación en XAI responde más a criterios de utilidad y simplicidad interpretativa que a una definición formal y universalmente aceptada.

En esta tesis nos centramos en *SHAP* [**?**], el cual es un método que, en principio, provee explicaciones agnósticas al modelo, con un alcance local y que es del tipo de feature attribution. Esto significa que para

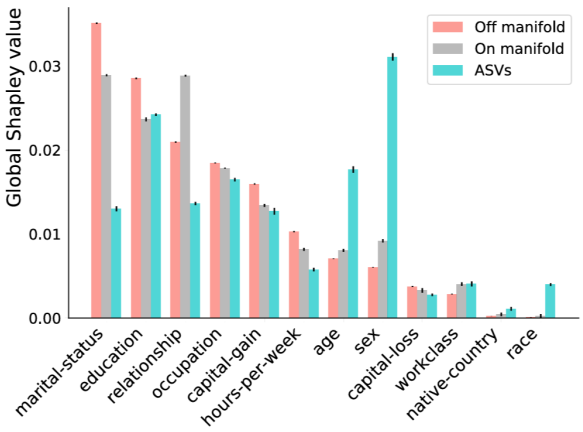


Figura 1: Valores globales de los Shapley Values y ASV para el modelo entrenado para el Census Income dataset.

una predicción individual que realiza el modelo, *SHAP* va a devolver un valor asociado a cada feature de la instancia. Decimos que es agnóstico, puesto que su valor depende meramente del ouput del modelo, por lo que no esta acoplado a su representación interna. Aún así, *SHAP* actualmente es un framework que engloba varios tipos de métodos, ya que hay distintas aproximaciones según cuál sea tu tipo de modelo. Por ejemplo, existen métodos como DeepExplainer, TreeExplainer y LinearExplainer dentro del framework SHAP, los cuales se utilizan para modelos específicos, pero también hay otros métodos como KernelExplainer, que son agnósticos al modelo.

Uno de los problemas principales de este framework es lo costoso que es calcularlo, y en particular, hay estudios que analizan esta dificultad desde el punto de vista de la complejidad computacional. Por ejemplo, se demostró que calcular SHAP para un clasificador trivial era NP-completo, o intratable, para datos con distribuciones más o igual de complejas que una Naive Bayes [**?**]. A raíz de esto, nos resultó interesante analizar si existía alguna variación de SHAP, la cual pueda ser calculada en tiempo polinomial para una distribución de red bayesiana. Además, actualmente hay trabajos similares a este, que buscan analizar la complejidad de SHAP según la variante, el modelo y la distribución [**?**]

SHAP está basado en los Shapley values [**?**], un valor de teoría de juegos. Una variación de los mismos son los assymetric shapley values (ASV), los cuales pueden introducir el factor de causalidad al cálculo de los mismos. De esta manera, se define ASV [**?**], un framework similar a SHAP, el cual además tiene en cuenta el grafo causal de los datos. El objetivo es esta tesis es lograr calcular ASV en tiempo polinomial de forma exacta y aproximada para datos con distribuciones de redes bayesianas.

1.1. Explicaciones basadas en causalidad: Un ejemplo comparativo entre SHAP y ASV

Echu: ¿Juega el ejemplo así?

En esta sección presentamos un ejemplo tomado de la sección 4.1 del artículo original de ASV [**?**], con el objetivo de ilustrar como la incorporación de información causal puede enriquecer las explicaciones de un modelo. Se utiliza como caso de estudio datos del conjunto Census Income de la UCI [**?**], en el cual se entrena un modelo (en este caso una red neuronal) para predecir si el ingreso de un individuo supera los $50 000. Dado que algunas de las variables demográficas presentes en este conjunto (por ejemplo, la edad) son claramente causas de otras (por ejemplo, el nivel educativo), es especialmente pertinente considerar información causal al interpretar la predicción del modelo. El conjunto A son las variables definidas como los ancestros causales, con $A = \{age, sex, native\ country, race\}$ y el resto de variables son descendientes de estas, dentamos a los descendientes como el conjunto D.

Para este experimento se calculan dos variaciones de SHAP, una *off-manifold* y otra *on-manifold*, la diferencia es la función de probabilidad que ambas usan. En el enfoque *on-manifold* se respetan las restricciones que hay entre las distintas variables, por ejemplo a la hora de calcular SHAP se va a descartar cualquier instancia tal que $age = 3 \wedge occupation = teacher$, ya que eso no es posible. Pero en el enfoque *off-manifold* no va a haber ninguna restricción a la hora de combinar los distintos valores de cada feature.

La Figura 1 muestra como la variable *género* presenta un Shapley value relativamente pequeño en el análisis off-manifold. En cambio, al incorporar el conocimiento causal mediante ASV, se observa que

dicha variable recibe un valor significativamente mayor. Este hallazgo evidencia que, a pesar de que el género tenga una influencia moderada en el modelo cuando se considera de forma aislada, su papel en la explicación se amplifica una vez que se tiene en cuenta su relación causal con otras variables (como el estado civil o la relación actual). De esta forma, los ASVs proporcionan una medida más precisa y contrastada de la contribución de cada variable, permitiendo detectar aspectos de discriminación o sesgo que de otro modo pasarían desapercibidos.

Este ejemplo evidencia cuándo es útil incorporar información causal en la explicación de modelos: en situaciones donde se conoce una relación de causa y efecto entre las variables, el uso de ASVs no solo mejora la interpretabilidad de la explicación, sino que también aporta una perspectiva más fiel al proceso generador de los datos, contribuyendo así a la construcción de modelos más transparentes y confiables.

1.2. Código fuente

Todos los algoritmos presentados en este trabajo fueron implementados. El código fuente puede ser encontrado online en el siguiente repositorio:

Repository	**Repository**
Source code	https://github.com/EchuCompa/pasantia-BICC

2. Introducción a Shapley y ASV

2.1. SHAP

Sea X un conjunto finito de features. Una entidad e sobre X es una función $e : X \to \{0, 1\}$, tal que para una feature x, $e(x)$ indica el valor que la entidad e toma en x. Utilizamos clasificadores binarios y features binarios para esta definición, ya que esta restricción nos permite simplificar la notación y las ideas presentadas, sin perder generalidad en los resultados. En particular, las técnicas y algoritmos que desarrollamos en este trabajo pueden extenderse naturalmente al caso de clasificadores multiclase. El dominio de las instancias que va a tomar nuestro clasificador lo denotamos como $\texttt{ent}(X)$ [1], el cual es el conjunto de todas las posibles $2^{|X|}$ entidades. El espacio de probabilidad para el conjunto $\texttt{ent}(X)$ va a estar dado por Pr. Así es como podemos definir a un clasificador binario $M : \texttt{ent}(X) \to \{0, 1\}$ sobre entidades [2] como una función, la cual dada una entidad e, $M(e)$ indica la clase asignada por el clasificador a e.

Un *feature attribution score* para un modelo M y una entidad e es una función $\phi : X \to \mathbb{R}$, tal que $\phi(x)$ indica el *puntaje* o *relevancia* del feature x con respecto a la predicción $M(e)$. Uno de los puntajes de feature attribution más destacados es él SHAP-SCORE [**?**], que se basa en los Shapley values [**?**] de la teoría de juegos cooperativos. En ese contexto, los Shapley values representan el esquema único de distribución de la ganancia obtenida por una coalición de jugadores que satisface ciertas propiedades deseables. Una interpretación de los mismos es una función que nos dice cuánto aporta cada jugador al valor total que obtiene la coalición.

Más formalmente, sea $\mathcal{I}$ un conjunto finito de jugadores, y definimos una *función característica* para $\mathcal{I}$ como una función $\nu : \mathcal{P}(\mathcal{I}) \to \mathbb{R}$, que asigna un valor a cada posible *coalición* de jugadores (es decir, subconjuntos de los jugadores). Por ejemplo, si los jugadores son features, se podría tomar $v(S)$ como la predicción promedio del modelo cuando los features de S se dejan fijos con ciertos valores. Esta valuación daría un valor mayor en la medida que los valores fijos para las features de S estén más correlacionados con que el modelo acepte. Los Shapley values $\{\phi_i\}_{i\in\mathcal{I}}$ son las únicas funciones que toman como entrada funciones características y devuelven valores reales que satisfacen las siguientes propiedades:

- **Eficiencia**: toda la ganancia es distribuida.

$$\sum_{i\in\mathcal{I}} \phi_i(\nu) = \nu(\mathcal{I})$$

[1] Podríamos considerar un codominio no binario pero finito $\mathbb{D}_x$ para cada $x \in X$ y adaptar todas las definiciones
[2] Aquí también podríamos considerar modelos con un codominio finito

- **Simetría**: cualquier par de jugadores $i, j \in \mathcal{I}$ que contribuyan igual reciben la misma recompensa.

$$\forall i, j \in \mathcal{I} : \left(\bigwedge_{S \subseteq \mathcal{I} \setminus \{i,j\}} \nu(S \cup \{i\}) = \nu(S \cup \{j\}) \right) \implies \phi_i(\nu) = \phi_j(\nu)$$

- **Linealidad**: si dos juegos se combinan, entonces la solución a ese nuevo juego es la suma de las soluciones de los originales. Si un juego es multiplicado por un escalar, entonces la solución también se multiplica por él.

$$\forall a \in \mathbb{R} : \phi_i(a\nu_1 + \nu_2) = a\phi_i(\nu_1) + \phi_i(\nu_2)$$

- **Jugador nulo**: si un jugador no contribuye en ninguna coalición, entonces no recibe recompensa.

$$\forall i \in \mathcal{I} : \left(\bigwedge_{S \subseteq \mathcal{I} \setminus \{i\}} \nu(S) = \nu(S \cup \{i\}) \right) \implies \phi_i(\nu) = 0$$

Además, existe una forma cerrada para estas funciones. Dado un conjunto finito A, sea $perm(A)$ el conjunto de todas sus permutaciones[3], y dada $\pi \in perm(A)$ denotamos como $\pi_{<a}$ al conjunto $\{a' \in A : \pi(a') < \pi(a)\}$. Entonces:

$$\phi_i(\nu) = \frac{1}{|\mathcal{I}|!} \sum_{\pi \in perm(\mathcal{I})} [\nu(\pi_{<i} \cup i) - \nu(\pi_{<i})] \tag{1}$$

Intuitivamente, esta función considera todos los órdenes posibles en que los jugadores llegan al juego y utiliza la contribución que i proporciona al llegar. Se puede demostrar que:

$$\phi_i(\nu) = \sum_{S \subseteq \mathcal{I} \setminus \{i\}} c_{|S|} \left[\nu(S \cup i) - \nu(S)\right]$$

donde $c_m = \frac{m!(|\mathcal{I}|-m-1!)}{|\mathcal{I}|!}$.

La analogía con el aprendizaje automático surge al entender el conjunto de n features X como jugadores, y la función característica ν como la predicción promedio al considerar un subconjunto de estos features fijados. En el ejemplo que vimos en la subsección 1.1, X serían los features del dataset (age, sex, etc.) y ν sería la predicción promedio que utiliza a la red neuronal, que evalúa si el ingreso de una persona es superior a \$50 000 para realizar la predicción. Dados M y e, definimos el conjunto de entidades consistentes con (*consistent with*) e teniendo en cuenta el subconjunto de features $S \subseteq X$ como $\texttt{cw}(e, S) = \{e' \in \texttt{ent}(X) : e'(s) = e(s) \text{ para } s \in S\}$. Definimos a la probabilidad condicionada como:

$$\Pr\big[e' \mid \texttt{cw}(e, S)\big] = \begin{cases} \dfrac{\Pr[e']}{\sum\limits_{e'' \in \texttt{cw}(e,S)} \Pr[e'']} & \text{si } e' \in \texttt{cw}(e, S), \\ 0 & \text{en caso contrario.} \end{cases}$$

De este modo se define la función característica como:

$$\nu_{M,e,\Pr}(S) = \sum_{e' \in \texttt{cw}(e,S)} \Pr\big[e' \mid \texttt{cw}(e, S)\big]\ M(e'). \tag{2}$$

Por conveniencia, para un modelo M, una entidad e y una distribución Pr, denotaremos los Shapley values cómo:

$$Shap_{M,e,\Pr}(x_i) = \sum_{S \subseteq X \setminus \{x_i\}} c_{|S|} \left[\nu_{M,e,\Pr}(S \cup \{x_i\}) - \nu_{M,e,\Pr}(S)\right]$$

Nótese que los axiomas que estos valores satisfacen no tienen un significado claro en el contexto de la inteligencia artificial, ya que dependen de la definición de $\nu_{M,e,\Pr}$ [**?**]. Además, para algunas nociones simples y robustas de *feature attribution* basadas en *explicaciones abductivas* [**?**], los Shapley values no logran asignar un puntaje de 0 a features irrelevantes [**?**].

[3] Formalmente, $perm(A) = \{(a_1, a_2, \ldots, a_n) \mid \{a_1, a_2, \ldots, a_n\} = A\}$, es decir, el conjunto de todas las secuencias que se pueden formar reordenando los elementos de A.

2.2. Complejidad de las explicaciones basadas en Shapley values

2.2.1. Resultados conocidos sobre los Shapley Values

Calcular estos valores en tiempo polinomial con respecto al tamaño del modelo es un desafío: por ejemplo, la sumatoria externa itera sobre un conjunto de tamaño exponencial en el número de features n. Sin embargo, para algunas familias específicas de modelos y distribuciones, es posible desarrollar algoritmos eficientes.

El primer resultado de este tipo provino de [**?**], donde los autores proporcionan un algoritmo en tiempo polinomial para calcular los Shapley values en árboles de decisión bajo la distribución *producto* o *completamente factorizada*. Tal distribución surgiría naturalmente bajo el supuesto poco realista de *independencia de features*. En dicho escenario tendríamos, para cada $x \in X$, un valor p_x que indica la probabilidad de que el feature x tenga valor 1 en una entidad aleatoria. Así, se sigue que:

$$\Pr[e'|\mathtt{cw}(e,S)] = \prod_{\substack{x\in X\setminus S\\ e'(x)=1}} p_x \prod_{\substack{x\in X\setminus S\\ e'(x)=0}} (1-p_x)$$

Estos resultados se extendieron en [**?**], donde se demostró que también es posible calcular los Shapley values para distribuciones producto cuando el modelo está condicionado a ser un circuito *determinístico* y *descomponible*. Además, se mostró que eliminar cualquiera de estas condiciones hace que el problema sea #P-HARD, y en un artículo posterior también obtuvieron resultados de no-aproximabilidad [**?**].

Un resultado más general fue demostrado simultáneamente en [**?**]: es posible calcular los Shapley values para una familia de modelos $\mathcal{F}$ bajo la distribución producto si y solo si es posible calcular la predicción promedio para ese modelo dado cualquier conjunto de probabilidades $\{p_x\}_{x\in X}$ en tiempo polinomial. A través de este lema, deducen inmediatamente la factibilidad de calcular los Shapley values para modelos de regresión lineal, árboles de decisión, funciones booleanas d-DNNF y circuitos CNF de anchura acotada. Luego, también demostraron la intratabilidad de este problema para modelos más expresivos como modelos de regresión logística, redes neuronales con funciones de activación sigmoide y funciones booleanas generales en CNF.

En [**?**] se afirma que es posible calcular los Shapley values para árboles de decisión bajo la *distribución empírica*, que es la dada por los datos de entrenamiento. Más formalmente, dado un multiconjunto de muestras $D \subseteq \mathtt{ent}(X)$ de tamaño m, la distribución empírica inducida por D se define como:

$$\Pr[e'] = \frac{D(e')}{m}$$

donde $D(e')$ indica el número de copias de e' que contiene D. Observar que la probabilidad de una entidad no vista es 0.

Sin embargo los autores no proporcionan una demostración que respalde la corrección del algoritmo y, además, en [**?**] se demuestra que, para este tipo de distribución, el problema de calcular los Shapley values es #P-HARD incluso para modelos extremadamente simples[4], y en particular, para árboles de decisión.

2.2.2. Asymmetric Shapley Values (ASV)

La definición de los Shapley values en la Ecuación 1 asigna el mismo peso a todas las posibles permutaciones. En general, podríamos considerar una función de peso $w : perm(\mathcal{I}) \to R$ y definir

$$\phi_i^{assym}(\nu) = \sum_{\pi\in perm(\mathcal{I})} w(\pi)\,[\nu(\pi_{<i}\cup i) - \nu(\pi_{<i})] \tag{3}$$

Asumiendo $\sum_{\pi\in perm(\mathcal{I})} w(\pi) = 1$, esta es la expresión más general para cualquier función que satisfaga Eficiencia, Linealidad y Jugador Nulo [**?**]. Para cualquier función de peso diferente de la uniforme, ϕ_i^{assym} no satisface Simetría (y de ahí el nombre).

En [**?**], se definen los *Asymmetric Shapley Values* considerando la definición de la Ecuación 3 y una función de peso basada en el grafo causal del espacio de features. Más formalmente, se asume que tenemos acceso a un DAG (Directed Acyclic Graph) $G = (X, E)$, donde los nodos de G son los features X. El

[4] Más precisamente, la afirmación de dificultad se aplica a cualquier familia de modelos que contenga funciones dependientes de solo uno de los features de entrada

conjunto $topos(G)$ de órdenes topológicos de G es un subconjunto de $perm(X)$, y podemos definir una función de peso w de la siguiente manera[5]):

$$w(\pi) = \begin{cases} \frac{1}{|topos(G)|} & \pi \in topos(G) \\ 0 & \text{en otro caso} \end{cases}$$

y los *Asymmetric Shapley Values* como:

$$Shap^{assym}_{M,e,\Pr}(x_i) = \frac{1}{|topos(G)|} \sum_{\pi \in topos(G)} [\nu_{M,e,\Pr}(\pi_{<i} \cup \{x_i\}) - \nu_{M,e,\Pr}(\pi_{<i})]$$

Intuitivamente, los Asymmetric Shapley Values filtran permutaciones que no respetan la causalidad definida por el DAG G. En el ejemplo que vimos en 1.1, una permutación que respeta la causalidad sería $topos$, tal que $topos[\text{edad}] < topos[\text{educación}]$, ya que $(\text{edad}, \text{educación})$ es un eje de G. Nos importan estas permutaciones ya que queremos evaluar la mejora del modelo sabiendo la edad, y luego cuánto mejora el modelo si conocemos la educación, *además* de la edad. De esta forma si la educación queda fija, la edad también, por lo que *ASV* va a terminar asignando *más importancia a las causas*, ya que las evalúa primero, y *menos importancia a las consecuencias*, puesto que las evalúa cuando la causa ya fue incluida. Esto resulta deseable en contextos explicativos, donde típicamente nos interesa priorizar las variables que originan un fenómeno, y no aquellas que simplemente son consecuencia del mismo.

Este grafo causal se introdujo para modelar correlaciones entre las variables a nivel del puntaje mismo, independientemente de la distribución subyacente. Sin embargo, podemos considerar que, en lugar de un grafo causal, se nos proporciona una Red Bayesiana que describe la distribución del espacio de features, la cual en particular contiene un DAG que podemos emplear como grafo de causalidad. Lo que estamos haciendo es *tomar a la red bayesiana como nuestro digrafo causal*. Esto es clave a la hora de tener en cuenta los distintos experimentos realizados en este trabajo, puesto que tenemos redes bayesianas pero no digrafos causales. Aun así, se podría introducir un grafo causal distinto como input y los algoritmos presentados en las secciones siguientes funcionarían de igual manera.

En [**?**], se demostró que calcular los Shapley values para una Naive Bayes es NP-HARD, al considerar el modelo trivial $f(x_1, \ldots, x_n) = x_1$. Lo que esto nos dice es que calcular SHAP para una distribución y un modelo de poca complejidad ya resulta intratable. Una Naive Bayes es una red cuyo DAG tiene forma de estrella: hay un único nodo x_1 tal que el conjunto de aristas es $E = \{(x_1, x_j) : 2 \leq j \leq n\}$ (x_1 es padre de todos los nodos, y no hay otras aristas, como se puede ver en la Figura 2).

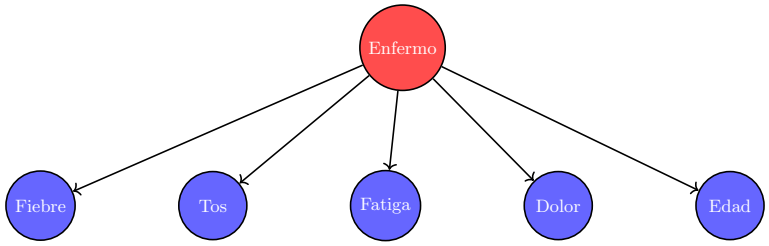


Figura 2: Distribución Naive Bayes para predicción de enfermedades. La variable **Enfermo** influencia al resto.

Para calcular la probabilidad de una entidad e en una Naive Bayes, lo que hacemos es calcular la probabilidad de x_1 (la raíz) y luego la probabilidad del resto de los nodos condicionados en el valor x_1. La fórmula la podemos ver a continuación:

$$\Pr[e] = \Pr[X_1 = e(x_1)] \prod_{j=2}^{n} \Pr[X_j = e(x_j) | X_1 = e(x_1)]$$

Teniendo en cuenta que nuestra distribución es una Naive Bayes, calcular los Asymmetric Shapley Values (considerando la red misma como el DAG de causalidad) puede hacerse en tiempo polinomial para una familia de modelos más grande, a diferencia de los Shapley values usuales. Más específicamente, para cualquier modelo que permita calcular los Shapley values normales para la distribución producto:

[5]Es un problema #P-HARD calcular $|topos(G)|$ para cualquier DAG [**?**], pues un orden topológico es una extensión lineal. Pero para algunas familias de dígrafos es posible calcularlo en tiempo polinomial, como los polytrees con grado acotado, como vamos a ver en la sección 6.3

Teorema 1. *Los Asymmetric Shapley Values pueden calcularse en tiempo polinomial para distribuciones dadas como una Red Bayesiana Naive y para una familia de modelos $\mathcal{F}$ si y solo si los Shapley values pueden calcularse para la familia $\mathcal{F}$ bajo una distribución producto arbitraria en tiempo polinomial.*

La demostración de este teorema se puede encontrar en el apéndice en la sección 10.2.1.

A raíz del resultado obtenido con el Teorema 1, nos gustaría encontrar modelos en los cuales se pueda calcular la predicción promedio en tiempo polinomial, puesto que si no podemos calcular el promedio en tiempo polinomial, es razonable pensar que calcular ASV tampoco resultaría tratable, ya que en principio habría que evaluar al promedio. Por lo tanto, decidimos enfocarnos puntualmente en los árboles de decisión. En la siguiente sección vamos a introducir más formalmente a las redes bayesianas y cómo calcular el promedio para modelos del tipo *Decision Trees*, con datos que tienen como su distribución a una red bayesiana, siendo esta también su grafo causal.

3. Grafos Causales

Como mencionamos en la sección anterior, *ASV* utiliza el grafo causal asociado al problema para definir una función w, que nos va a permitir filtrar permutaciones no deseadas que no respeten la causalidad. Luego, cuando tenemos una permutación *topos* que sí nos interesa, vamos a querer realizar la operación $\nu(\pi_{<i})$, la cual consiste en evaluar la esperanza teniendo en cuenta nuestra distribución elegida y un valor fijo para los features en $\pi_{<i}$. Para realizar este cálculo necesitamos entender cómo se calculan probabilidades en una red bayesiana y cuál es su complejidad.

3.1. Redes Bayesianas

Una Red Bayesiana N es un *DAG*, donde cada nodo representa una variable diferente y los arcos representan las dependencias condicionales entre ellas, puesto que para calcular la probabilidad de la cabeza necesitamos la de la cola. Por ejemplo, si tenemos el arco $A \longrightarrow B$, esto representa que la variable B depende[6] de A, y esto se cuantifica usando la probabilidad condicional $P(B|A)$. Un nodo puede tener múltiples padres, por lo que la distribución de los valores del nodo se definirá en una *Tabla de Probabilidad Condicional* (CPT), que define la probabilidad de que tome algún valor, dado los posibles valores de sus nodos padres. Con esto, podemos definir una Red Bayesiana como:

Definición 3.1. Una Red Bayesiana para variables X es una tupla $N = (X, E, \Pr)$, donde (X, E) es un DAG que tiene las variables X como nodos, E como aristas y Pr es una función que codifica, para cada variable $x \in X$, su distribución de probabilidad condicional $\Pr(x|Parents(X))$:

- Para cada variable $x \in X$, sabemos que x tiene un conjunto finito de estados mutuamente excluyentes. Estos son los posibles valores que puede tomar.
- Para cada variable $x \in X$ con padres $B_1, \ldots, B_n$, se tiene una tabla de probabilidad condicional (CPT) $Pr(x \mid B_1, \ldots, B_n)$. Así es como está definida Pr.

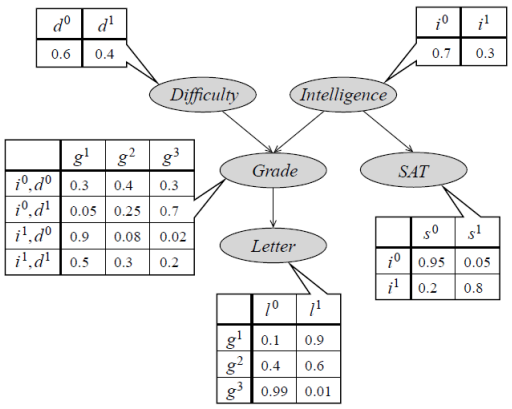


Figura 3: Red Bayesiana para determinar la inteligencia de un estudiante. Podemos ver que cada nodo tiene una CPT que se calcula en función de sus padres. Aquí las variables son ternarias o binarias. Fuente: [**?**]

Este tipo de redes nos ayuda a simplificar el cálculo de distribuciones de probabilidad conjunta. Una de sus propiedades más importantes es la *regla de la cadena*, que proporciona una estructura fundamental para entender cómo las probabilidades conjuntas pueden descomponerse en estas redes. Específicamente, establece que si consideramos que N es una red Bayesiana sobre un conjunto de variables $\{A_1, \ldots, A_n\}$, entonces N especifica una única distribución de probabilidad conjunta $Pr(U)$, donde $U = \{A_1, \ldots, A_n\}$, que puede expresarse como el producto de todas las CPT's especificadas en N. Matemáticamente, se representa como:

$$Pr(U) = \prod_{i=1}^{n} P(A_i \mid Parents(A_i)) \tag{4}$$

[6] Aunque no necesariamente significa que A dependa de B causalmente, puesto que podríamos armar una red bayesiana que no se condiga con el DAG causal de las variables.

donde $Parents(A_i)$ denota los padres de A_i en la red. Esta regla es esencial porque simplifica el cálculo de la distribución de probabilidad conjunta al descomponerla en dependencias locales más simples entre un nodo y sus predecesores directos. Intuitivamente, lo que hace es utilizar las dependencias locales para definir una distribución global, facilitando los procesos de inferencia dentro de estas redes.

Además, las redes Bayesianas contemplan una variedad de operaciones computacionales cruciales para el razonamiento probabilístico. Una de las operaciones principales es la **inferencia**, que implica calcular la distribución a priori para alguna variable. Es decir, calcular $Pr(A \mid B_1 \dots B_i)$, siendo A un nodo en la red y B_i otros nodos de la red. Este proceso a menudo se denomina *consulta* a la red. La complejidad de la inferencia depende de la estructura de la red; para redes con forma de polytree (DAG's que su grafo subyacente es un árbol), también conocidas como *simplemente conectadas*, la complejidad es polinomial en el tamaño de las variables de la red [**?**], pero para redes generales la inferencia es #P-HARD.

El algoritmo que se utiliza para realizar la inferencia marginal, que calcula cual es la probabilidad de que una variable tome ciertos valores, es *Variable Elimination* [**?**]. Su complejidad depende principalmente del treewidth[7] de la red y puntualmente para redes de un treewidth acotado, su complejidad es polinomial respecto al tamaño de la red. En base a esto es que elegimos trabajar con polytrees.

3.2. Predicción promedio en árboles de decisión

Los árboles de decisión son ampliamente utilizados en Inteligencia Artificial (IA) debido a su capacidad para realizar predicciones y clasificaciones basadas en features de los datos de entrada. Al descomponer decisiones en una serie de preguntas y respuestas simples, los árboles de decisión permiten a los usuarios entender el razonamiento detrás de las predicciones del modelo, contribuyendo a la transparencia en sistemas complejos [8].

Definición 3.2. Un **árbol de decisión** es una estructura jerárquica utilizada para representar funciones de decisión sobre un conjunto finito de variables. Formalmente, un árbol de decisión T sobre un conjunto de variables X es un árbol enraizado cuyas componentes principales son:

- **Nodos internos**, cada uno asociado a una variable $x \in X$ y etiquetado con una condición sobre dicha variable. Estos nodos determinan cómo se bifurcan los datos.
- **Ramas**, que conectan un nodo padre con sus hijos y representan los posibles valores o resultados de evaluar la condición del nodo padre.
- **Hojas**, que son nodos sin hijos y contienen una salida concreta: una clase, un valor numérico, o una distribución de probabilidad, dependiendo del tipo de problema (clasificación, regresión, o probabilístico).

Cada instancia $e \in \texttt{ent}(X)$ se evalúa recorriendo el árbol desde la raíz hasta una hoja, tomando decisiones en función de las condiciones de los nodos internos. El resultado asociado a la hoja alcanzada es la predicción del modelo para esa instancia.

Como mencionamos previamente, investigamos si el cálculo del promedio era tratable para árboles de decisión con distribuciones de redes bayesianas, ya que nuestro objetivo era calcular ASV en tiempo polinomial.

Para este algoritmo[9] trabajaremos con árboles de decisión binarios y con features binarios, más adelante veremos la extensión a variables no binarias. Cada nodo determinará un valor para un feature X_i, donde el hijo izquierdo corresponde al caso $X_i = 0$ y el hijo derecho al caso $X_i = 1$. Los features son un conjunto X, y su distribución de probabilidad será una red bayesiana $N = (V, E, Pr_B)$. Definimos a ev y $pathCondition$ cómo conjuntos de asignaciones $X_i = k$, las cuales definen que el feature X_i toma el valor k. Entonces, $Pr_B(pathCondition \mid ev)$ representa la probabilidad de que dada una instancia que tiene los valores de ev, la instancia tome los valores de $pathCondition$, dada la red bayesiana N. La idea del algoritmo consiste en explorar todas las ramas e ir acumulando las decisiones tomadas en la variable $pathCondition$. Al llegar a la hoja evaluamos la probabilidad de haber llegado a la hoja dada la evidencia y la multiplicamos por el valor que la misma devuelve.

[7] El treewidth (ancho de árbol) de un grafo es una medida de cuán "cercano´ ´ está un grafo a ser un polytree. Formalmente, es el tamaño del mayor conjunto de vértices en una descomposición en árbol, menos uno, minimizado sobre todas las posibles descomposiciones.

[8] Aunque no siempre el camino de un árbol es la mejor explicación, ya que puede haber features redundantes en el camino que no sean claves para la predicción realizada. [**?**]

[9] Pueden encontrar este algoritmo en `\pasantia-BICC\asvFormula\bayesianNetworks\bayesianNetwork.py`

Algorithm 1 Predicción promedio para árbol de decisión binario

function MEAN($node$, B, $pathCondition$, $evidence$)
  if $evidence$ does not match $pathCondition$ **then**
    return 0
  end if
  if $node$.isLeaf **then**
    return $Pr_B(pathCondition \mid evidence) \cdot node.value$
  end if
  $X_i \leftarrow node.feature$
  $leftMean \leftarrow$ MEAN($node$.left, B, $pathCondition \cup \{X_i = 0\}, evidence$)
  $rigthMean \leftarrow$ MEAN($node$.right,B,$pathCondition \cup \{X_i = 1\}, evidence$)
  return $leftMean + rigthMean$
end function

Analicemos la complejidad del Algoritmo 1. La condición del primer `if` puede ser evaluada en tiempo $O(|V|)$. En las hojas solo hacemos un llamado al algoritmo de *variable elimination*, cuya complejidad denotamos como $O(varElim)$. Por otro lado, en los nodos internos solo se hacen los llamados recursivos extendiendo la *pathCondition*, lo cual puede implementarse en $O(1)$. Por lo tanto, si l denota la cantidad de hojas de nuestro árbol de decisión, e i la cantidad de nodos internos, la complejidad de nuestro algoritmo es $O(i|V| + (varElim)l)$. En el caso de los polytrees $varElim$ es polinomial, por lo que nuestro algoritmo realizaría una cantidad de operaciones polinomial en función del tamaño del árbol de decisión, siendo estas $(O(i + l) = O(|V|))$ operaciones polinomiales.

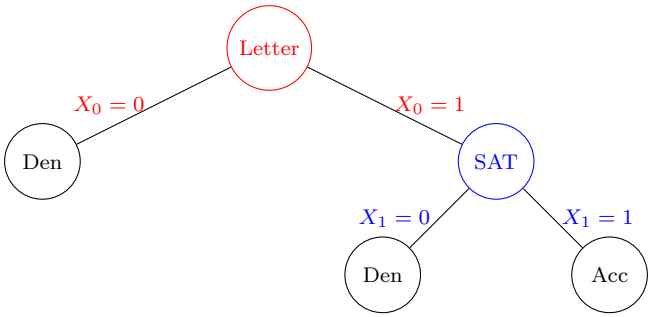


Figura 4: Decision tree que define si un estudiante va a ser aceptado (accepted/1) o rechazado (denied/0) en su ingreso a una universidad basado en los features de la red bayesiana de la Figura 3. Podría ser generado a partir de un dataset con los mismos features de la red, con un feature extra llamado `Acceptance`. La red usada puede encontrarse en el repositorio

En la Figura 4 tenemos un ejemplo de un árbol de decisión. Queremos ver cuál es la predicción promedio de nuestro árbol de decisión para un alumno que es inteligente. Si corremos nuestro algoritmo desde la raíz del árbol, lo que vamos a obtener es

$$
\begin{aligned}
&\text{MEAN}(Letter, B, \{\}, \{INT = 1\}) \\
&= \text{MEAN}(Den, B, Letter = 0, \{INT = 1\}) + \text{MEAN}(SAT, B, Letter = 1, \{INT = 1\}) \\
&= 0 + \text{MEAN}(SAT, B, \{Letter = 1\}, \{INT = 1\}) \\
&= \text{MEAN}(Den, B, \{Letter = 0, SAT = 0\}, \{INT = 1\}) + \text{MEAN}(Acc, B, \{Letter = 1, SAT = 1\}, \{INT = 1\}) \\
&= Pr_B(SAT = 0, Letter = 1 \mid INT = 1) \cdot 0 + Pr_B(SAT = 1, Letter = 1 \mid INT = 1) \cdot 1 \\
&= Pr_B(SAT = 1, Letter = 1 \mid INT = 1) \\
&= 0.61
\end{aligned}
$$

En este ejemplo podemos ver que aunque la inteligencia no sea una variable que se tenga en cuenta en el árbol, afecta la predicción promedio, ya que para calcularla estamos utilizando la red bayesiana, y esta evidencia introducida va a afectar la inferencia realizada en la red.

3.2.1. Expandiendo la predicción promedio a features no binarios

El Algoritmo 1 funciona para árboles binarios y para variables binarias. Para poder trabajar con features no binarios tuvimos que modificar la inferencia realizada, por lo que su complejidad dejó de ser polinomial en el tamaño de la red, ya que la implementación actual depende de la cardinalidad de cada feature.

Si cada feature admite más de un valor, cuando llegamos a un nodo n obtenemos su feature f y su umbral de decisión v. Luego para los valores $i, d \in Dominio(f)$ se le agregan a $pathCondition$ los i tal que $i < v$ en el lado izquierdo de la recursión y los d tal que $d \geq v$ en el lado derecho. Seguimos realizando la inferencia al llegar al nodo hoja a través de una suma de la unión de todas las consultas generadas, por lo que la inferencia no es polinomial. Por ejemplo, si llegamos con $pathCondition = \{x = \{1, 2\}, y = \{3\}\}$ vamos a tener que evaluar $Pr_B(pathCondition) = Pr_B(x = 1, y = 3) + Pr_b(x = 2, y = 3)$. Para mejorar esta inferencia, una posibilidad sería implementar la consulta modificando el algoritmo de Variable Elimination o creando nodos intermedios que representen la evidencia introducida; pero no tomamos este camino debido a que no es el objetivo principal de esta tesis.

4. Optimización para ASV : Clases de equivalencia

Recordemos la definición de la fórmula de ASV:

$$Shap^{assym}_{M,e,\Pr}(x_i) = \sum_{\pi \in topos(G)} [\nu_{M,e,\Pr}(\pi_{<i} \cup \{x_i\}) - \nu_{M,e,\Pr}(\pi_{<i})]$$

Para simplificar la notación, tomemos M, e y Pr fijos, de modo que $\nu_{M,e,\Pr} = \nu$. La idea es encontrar un criterio para disminuir el número de órdenes topológicos que necesitamos calcular, reduciendo la cantidad de veces que evaluaremos ν. La idea principal detrás de esta heurística es, identificar las clases de equivalencia para los diferentes órdenes topológicos $\pi^1, \pi^2 \in topos(G)$, tales que $\pi^1 R^\star \ \pi^2 \iff \nu(\pi^1_{<i}) = \nu(\pi^2_{<i})$. Nuestro algoritmo va a trabajar sobre una relación R más fina que $R^\star$, pues esta es costosa de calcular. Al conjunto de clases de equivalencia de G definido por la relación R lo vamos a denotar como $eqCl(G, x_i)$. Una vez que consigamos todas las clases de equivalencia $[\pi]_R$, solo necesitamos un representante de cada una y su tamaño para calcular el ASV:

$$\begin{aligned}
&Shap^{assym}_{M,e,\Pr}(x_i) \\
&= \frac{1}{topos(G)} \sum_{\pi \in topos(G)} \nu(\pi_{<i} \cup \{x_i\}) - \nu(\pi_{<i}) \\
&= \frac{1}{|topos(G)|} \sum_{[\pi]_R \in eqCl(G,x_i)} (\nu(\pi_{<i} \cup \{x_i\}) - \nu(\pi_{<i})) \cdot |[\pi]_R|
\end{aligned}$$

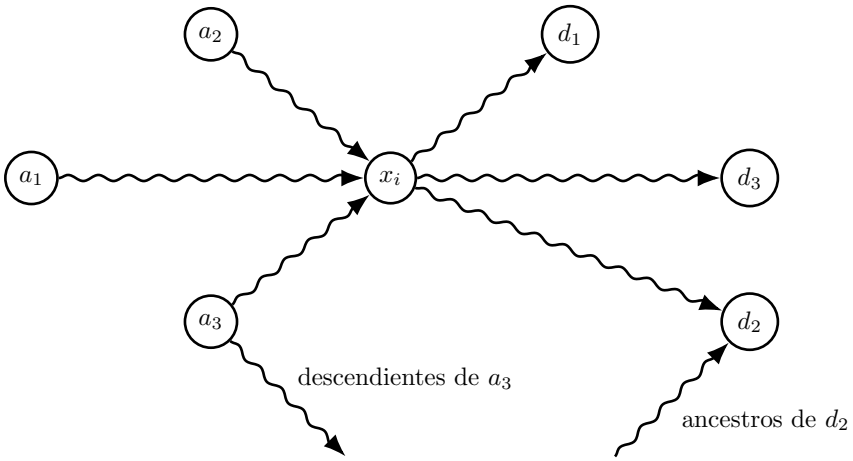


Figura 5: Al fijar un nodo x_i, podemos dividir el resto de los nodos en tres grupos: *ancestros* (todos los nodos que pueden alcanzar a x_i), *descendientes* (todos los nodos alcanzables desde x_i) y aquellos *no relacionados* con x_i. Los *no relacionados* son los que no pertenecen a los ancestros ni a los descendientes, por lo que pueden estar a la derecha o la izquierda de x_i en un orden topológico.

Sea nuestro DAG $G = (V, E)$, con A el conjunto de ancestros de x_i y D el conjunto de sus descendientes. Podemos ver estos conjuntos representados en la Figura 5. Para este ejemplo introductorio no hay ejes entre los ancestros y descendientes. Para cada orden topológico π, sabemos que todo $a \in A$ aparece antes de x_i, es decir $\pi(a) < \pi(x_i)$, y que todo $d \in D$ aparece después, $\pi(x_i) < \pi(d)$. Si esos fueran los únicos nodos a tener en cuenta, entonces $|topos(G)| = |A|! \cdot |D|!$, es decir, las permutaciones de A multiplicadas por las permutaciones de D. Por lo tanto, todos los órdenes tendrían los mismos features fijos antes de x_i, A, y los mismos después de x_i, D. Además, todos estarían en la misma clase de equivalencia definida por $R^\star$, pues el orden de los features en cada permutación π^1, π^2 *no afectará el resultado de evaluar* ν, $\nu(\pi^1) = \nu(\pi^2)$). Esto nos daría la siguiente fórmula:

$$\sum_{\pi \in topos(G)} [\nu(\pi_{<i} \cup \{x_i\}) - \nu(\pi_{<i})] = (\nu(\pi_{<i} \cup \{x_i\}) - \nu(\pi_{<i})) \cdot |A|! \cdot |D|!$$

Podemos utilizar cualquier $\pi \in topos(G)$ en este caso particular, puesto que todas las evaluaciones de $\nu(\pi_{<i})$ dan el mismo resultado para cualquier π (puesto que $\{\pi_{<i}\} = A$). Esto significa que podemos reducir el número de veces que evaluaremos ν, si logramos identificar estas clases de equivalencia. Teniendo en cuenta el caso en el cual tenemos nodos no relacionados, vamos a utilizar la relación R, en vez de $R^\star$, la cual tiene en cuenta a x_i, y se define cómo $\pi^1 R\ \pi^2 \iff \{\pi^1_{<i}\} = \{\pi^2_{<i}\}$. Esto nos dice que dos permutaciones están relacionadas por R si previo a la posición de x_i tienen el mismo conjunto de elementos. Los nodos que nos van a importar son los *no relacionados*, como podemos ver en la Figura 5, estos son los únicos que no van a tener un lugar respecto a x_i, definido previamente en base a su posición en el grafo.

Definición 4.1. Sea G un digrafo $G = (V, E)$. Una clase de equivalencia $[\pi]_R$ define en que posición respecto de x_i se encuentran los nodos de V, para un orden topológico π. Dada $f_\pi : V \setminus \{x_i\} \to \{left, right\}$, tal que $f_\pi(n)$ es una función que identifica si el nodo n está a la derecha o izquierda de x_i en π, la clase $[\pi]_R$ se ve representada como un conjunto de nodos con etiquetas $left, right$ según su orden:

$$\mathsf{Rep}([\pi]_R) = \{v_{f_\pi(v)} \mid v \in V \setminus \{x_i\}\}$$

$L([\pi]_R)$ y $R([\pi]_R)$ denotan el conjunto de nodos a la izquierda y a la derecha en la clase de equivalencia, respectivamente.

Definición 4.2. Sea G un digrafo $G = (V, E)$. Un orden topológico π' pertenece a la clase de equivalencia $[\pi]_R$ si se cumple que:

$$(\forall v \in V \setminus \{x_i\})(f_{\pi'}(v) = left \land v \in L([\pi]_R)) \lor (f_{\pi'}(v) = right \land v \in R([\pi]_R))$$

Esta es una relación más fina que $R^\star$, pues puede pasar que π^1 y π^2 no estén relacionadas en R pero sí en $R^\star$, por lo que separa en más clases. La ventaja de R es que no es necesario calcular ν para saber si dos elementos pertenecen a la misma clase, por lo que nos ahorra evaluaciones.

Si podemos calcular el número y tamaño de las clases de equivalencia de manera eficiente, entonces podemos calcular el valor de ASV. Para el tamaño de las clases de equivalencia, vamos a calcular el número de órdenes topológicos que pertenecen a la misma.

4.1. Número de órdenes topológicos de un DAG

Vamos a definir una fórmula para calcular el número de órdenes topológicos de un DAG, la cual va a ser útil para contar los tamaños de las clases de equivalencia. Queremos definir esto para ciertas familias de DAGs, porque sabemos que para el caso general es #P-HARD [**?**]. Comencemos con el DAG más básico, un grafo D con $r + 1$ nodos y sin aristas.

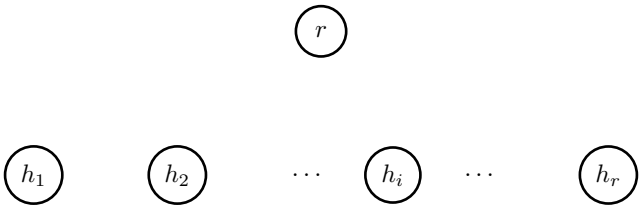


Figura 6: Digrafo vacío, sin aristas

En la Figura 6, el número de órdenes topológicos de D es $r + 1!$, porque los nodos no tienen ninguna arista entre ellos, por lo que no hay restricciones. Ahora añadamos algunas aristas a D para que se convierta en un árbol (dirigido) y agreguemos un subárbol debajo de cada nodo h_j. Así tenemos la Figura 7.

Aquí la fórmula es recursiva y depende de cada uno de los subárboles de D. Ahora tenemos $n + 1$ nodos, y los subárboles $t_1, \ldots, t_r$ de los hijos $h_1, \ldots, h_r$ tienen $k_1, \ldots, k_r$ nodos respectivamente (con $n = k_1 + \cdots + k_r$). Sea $\#topos(r)$ el número de órdenes topológicos del árbol D con raíz r. Entonces, la fórmula que tenemos es:

$$\#topos(t) = \binom{n}{k_1, k_2, \ldots, k_r} \prod_{i=1}^{n} \#topos(t_i)$$

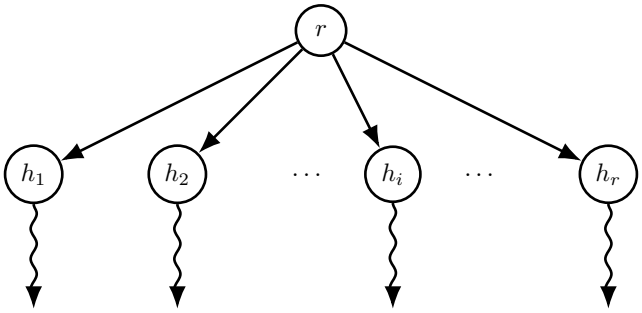


Figura 7: Polytree con nodos con grados de entrada menores o iguales a 1, lo cual definimos como *dtree*.

Podemos combinar los órdenes topológicos de cada hijo, seleccionando en qué posición asignamos a cada uno de ellos. El número de asignaciones diferentes que se pueden hacer en n posiciones con r conjuntos de k_i elementos cada uno es: $\binom{n}{k_1,k_2,\ldots,k_r} = \frac{n!}{k_1!k_2!\ldots,k_r!}$, el coeficiente multinomial. Ahora, para cada una de esas asignaciones, podemos usar cualquiera de los órdenes topológicos de cada subárbol; eso es él $\prod_{i=1}^{n} \#topos(t_i)$.

Podríamos intentar añadir más aristas a nuestro DAG D, por ejemplo, una arista (r, d) entre r y uno de sus descendientes d, como en la Figura 8.

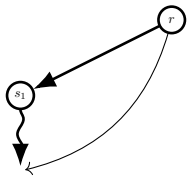


Figura 8: Arista entre descendiente de $s1$ y r

Pero dejaría de ser un polytree, pues introduciría un ciclo en el grafo no dirigido, y queremos limitarnos a contar los órdenes de este tipo de grafos. Algo que sí podemos añadir son múltiples raíces $r_1, \ldots, r_l$ en nuestro grafo, lo que seguiría siendo un polytree (o un polyforest). Y podemos calcular el #topos añadiendo una raíz *virtual* r_0 que esté conectada a todas ellas, para luego usar la misma fórmula como si fuera un árbol. Esto lo podemos observar en la Figura 9.

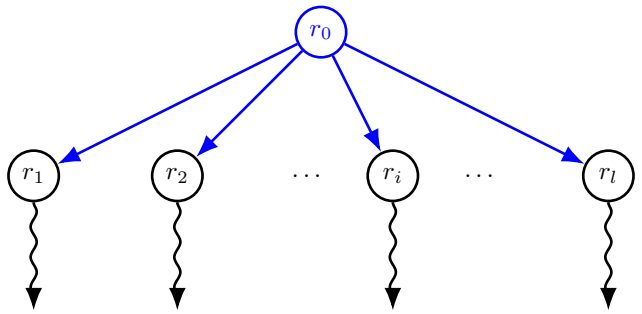


Figura 9: El DAG original son los nodos y aristas en negro, el nodo virtual y sus aristas están en azul.

Si tenemos múltiples raíces, entonces podría suceder que compartan algunos descendientes comunes. Pero si dos raíces r_i y r_j comparten dos o más descendientes d_1 y d_2, tales que hay dos caminos disjuntos de r_i a d_k y r_j a d_k, con $k \in \{1, 2\}$, entonces va a existir un ciclo en el grafo: $r_i \to d_1 \to r_j \to d_2 \to r_i$, por lo que no sería un polytree. Eso implica que dos raíces solo pueden compartir uno de estos nodos como máximo, puesto que sino el grafo subyacente va a tener un ciclo. Por ejemplo, este sería un polytree válido, como en la Figura 10.

En este caso, no podemos usar la misma fórmula que para el árbol, porque hay un solapamiento entre los descendientes de r_1 y r_2. Sabemos que no puede haber ninguna arista entre los subárboles de s_1, s_2 y s_3, porque eso crearía un ciclo. Para resolver este caso más general, polytrees, tuvimos que implementar una solución más compleja, entraremos más en detalle acerca de la misma en la sección 6. Por el momento,

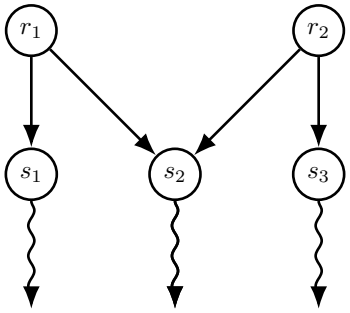


Figura 10: Ejemplo de polytree, para el cual no funciona la fórmula 4.3 para contar los órdenes topológicos en *dtrees*

simplemente vamos a trabajar con el caso resuelto previamente, el cual engloba polyforests con nodos con un padre como máximo. A estos los llamaremos *dtrees* (directed trees).

La fórmula que tenemos entonces es:

Definición 4.3. Dado un *dtree* t, con n subárboles t_i, cada uno de tamaño k_i, la cantidad de órdenes topológicos del mismo es:

$$\#topos(t) = \binom{n}{k_1, k_2, \ldots, k_r} \prod_{i=1}^{n} \#topos(t_i) \tag{5}$$

Analizando esta fórmula, podemos observar que:

- Para minimizar la cantidad de órdenes topológicos, es mejor tener una cantidad de subárboles pequeña.
- Si el grado máximo de salida m es lo suficientemente pequeño, entonces las combinaciones que vamos a realizar en el multinomial también van a ser pocas.
- La cantidad de órdenes está acotada por $n!$, y este es el caso en el cuál no tenemos ningún arista en el bosque.

4.2. Número de clases de equivalencias para *dtrees*

Con nuestra nueva definición de clase de equivalencia, la idea es calcular el número de clases de equivalencia de un DAG. Como se mencionó previamente, lo que vamos a analizar son los *nodos no relacionados* (unrelated) U y las distintas restricciones entre ellos. Estos son los nodos que no son descendientes ni ancestros de x_i (el feature para el cual estamos calculando el ASV). Comencemos con el caso de calcular el número de clases de equivalencia $[\pi]_R$ para un *dtree*con el feature x_i.

En el caso de la Figura 11, los nodos relevantes son los que están en rojo. Esto se debe a que los descendientes de x_i siempre estarán a la derecha de x_i en cualquier orden topológico y sus ancestros aparecerán a la izquierda, por lo que no definirán nuevas clases de equivalencia. Si no hubiera ningún eje en los nodos de U, entonces el número de clases sería $2^{|U|}$ [10], debido a que para definir un orden solo se necesita definir dónde insertar los nodos de U. Pero si no tienen ninguna arista que los conecte, entonces cada uno de ellos puede colocarse a la izquierda o a la derecha de x_i independientemente del resto, definiendo una nueva clase de equivalencia. Podemos observar que al no haber ejes entre los subárboles de b_1 y c_1, cada forma posible de ordenar los nodos de b_1 se puede mezclar con cada forma de c_1 (siempre respetando que c_1 aparezca luego de b_2). Gracias a esta observación es que podemos calcular los posibles órdeness topológicos y las respectivas clases de equivalencia de cada subárbol de nodos no relacionados, para luego combinar estos resultados. ¿Cómo podemos calcular estas clases de equivalencia, considerando solo los nodos del subárbol, para cada uno?

Para responder esta pregunta calculemos el número de clases de equivalencia para uno de los subárboles. Tomemos la Figura 12 como el subárbol b_1 del ejemplo anterior. Nuestro objetivo es calcular el número de clases de equivalencia del subárbol con raíz en b_1, $\#EC(b_1)$. Para b_1, tenemos dos opciones: puede ser

[10] Sólo queremos contabilizar las clases de equivalencia no vacías, por lo que deben tener al menos un orden topológico que pertenezca a la misma. Por lo que, por ejemplo, no vamos a tener en cuenta a clases con los ancestros de x_i a la derecha.

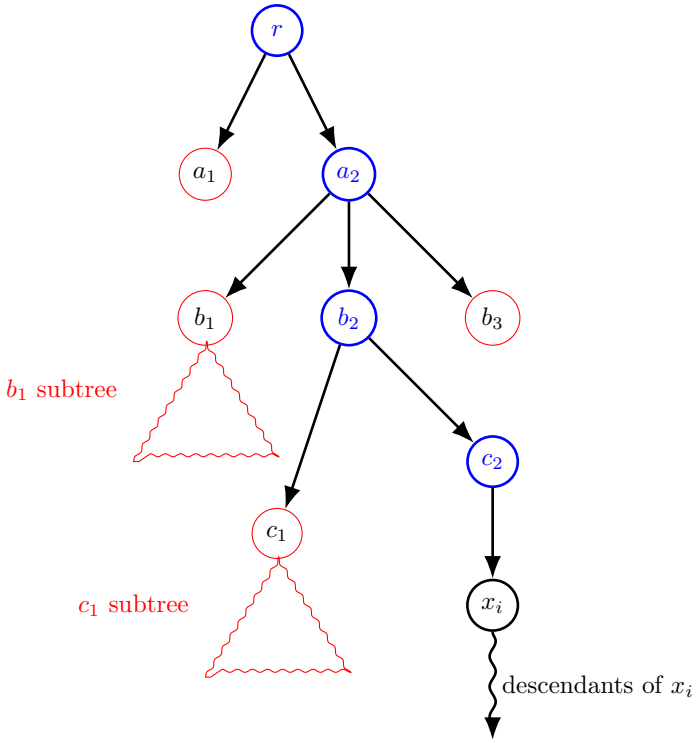


Figura 11: Ejemplo de un *dtree* con x_i en negro, sus nodos no relacionados marcados en rojo y sus ancestros marcados en azul.

posicionado a la derecha o a la izquierda de x_i en el orden topológico π. Si se posiciona a la derecha, entonces todos sus descendientes también estarán posicionados a la derecha, pues deben aparecer luego de b_1. Si se posiciona a la izquierda, entonces no hay ninguna restricción sobre sus descendientes. Entonces, nuestra fórmula sería:

$$\#EC(b_1) = \#EC(b_1|\pi(b_1) > \pi(x_i)) + \#EC(b_1|\pi(b_1) < \pi(x_i)) = 1 + \#EC(b_1|\pi(b_1) < \pi(x_i))$$

$\#EC(b_1|\pi(b_1) > \pi(x_i))$ lo sustituimos por 1, ya que los elementos a la izquierda de x_i van a ser los mismos para todos esos órdenes topológicos, puesto que sus descendientes van a estar a la derecha, por lo que todos los órdenes topológicos con b_1 a la derecha van a pertenecer a la misma $[\pi]_R$. Ahora, si b_1 está posicionado a la izquierda de x_i, entonces no hay restricciones para sus hijos ni sus subárboles, y podemos aplicar el mismo proceso para cada uno de sus hijos. Luego, podemos combinar cada $[\pi]_R$ obtenida en los subárboles, puesto que no tienen aristas en común. En términos de combinatoria, eso significa multiplicar el resultado que obtenemos para cada subárbol. También debemos tener en cuenta el caso en el que el nodo no tiene hijos; ahí podemos tener dos clases de equivalencia, considerando las posibilidades de izquierda y derecha. Así es como obtenemos esta fórmula:

Fórmula 1. *Dado un nodo n de un subárbol, la cantidad de clases de equivalencia con respecto a x_i se calcula como:*

$$\#EC(n) = \begin{cases} 2 & \text{si } n \text{ es una hoja} \\ \prod_{c \in children(n)} \#EC(c) + 1 & \text{oc.} \end{cases}$$

Esta fórmula también puede usarse para calcular las clases de equivalencia de todos los nodos *no relacionados* en el ejemplo anterior. Podemos usar una estrategia similar a la utilizada para calcular los órdenes topológicos. Creamos un nodo r_0 y lo conectamos a las raíces de todos los subárboles de los nodos no relacionados, y luego calculamos $\#EC(r_0)$ utilizando la fórmula definida previamente.

En la sección siguiente se presenta un algoritmo para obtener las clases de equivalencia, el cual es polinomial en la cantidad de clases. Por lo que utilizando la Fórmula 1, podemos prever el tiempo que va a tomar obtener las clases de equivalencias, puesto que ya tenemos una cota para las mismas. Esta fórmula la podemos correr en tiempo polinomial para realizar una aproximación del tiempo que va a tardar.

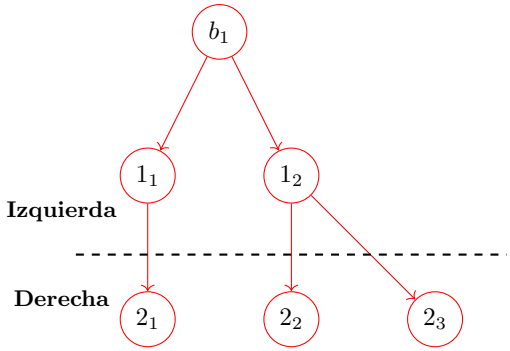


Figura 12: Ejemplo de una clase de equivalencia para los nodos del subárbol de b_1 en la Figura 11, que divide los nodos en aquellos ubicados a la izquierda o a la derecha de x_i en un orden topológico. En este caso la clase sería $L([\pi]_R) = \{b_1, 1_1, 1_2\}, R([\pi]_R) = \{2_1, 2_2, 2_3\}$

4.3. Cota superior para las clases de equivalencia

Queremos encontrar una cota superior para el número de clases de equivalencia de un árbol, para ver si el número de clases de equivalencia será menor que los posibles órdenes topológicos y si es que esta heurística trae consigo una mejora significativa.

Lema 1. *Sea T un árbol con su raíz n, donde l es el número de hojas del árbol y h es su altura. Entonces, para la fórmula:*

$$\#EC(n) = \begin{cases} 2 & \text{if } n \text{ is a leaf} \\ \prod_{c \in children(n)} \#EC(c) + 1 & \text{oc.} \end{cases}$$

tenemos la cota $\#EC(n) \leq h^l + 1$

La demostración está en el apéndice en la sección 10.2.2. Con esta cota podemos observar que:

- Si el número de hojas es $O(\log n)$ y la altura del árbol es $O(n)$, entonces el número de clases de equivalencia está acotado por $O(n^{\log n})$. Esta cota es *subexponencial*.
- Si la cantidad de hojas es pequeña, entonces nuestra cota va a disminuir también. Lo que significa que es mejor tener un pequeño grado de salida desde cada vértice, al igual que con los órdenes topológicos.
- Es mejor tener una mayor altura que un grado de salida mayor con mucha ramificación.
- La cota para el número de clases de equivalencia depende de h y l, a diferencia de la cota para los órdenes topológicos que solo depende de n, $O(n!)$. Más adelante veremos la diferencia en la práctica.

5. Algoritmo exacto para clases de equivalencia en *dtrees*

Nuestro objetivo es computar $Shap^{assym}_{M,e,\Pr}(x_i)$ dado un grafo causal G (que es un DAG). Recordemos que:

$$Shap^{assym}_{M,e,\Pr}(x_i) = \frac{1}{|topos(G)|} \sum_{[\pi]_R \in eqCl(G,x_i)} (\nu(\pi_{<i} \cup \{x_i\}) - \nu(\pi_{<i})) \cdot |[\pi]_R|$$

Una forma de obtener este valor es calculando primero el conjunto de clases de equivalencia $eqCl(G, x_i)$, para luego tomar un representante de cada una (es decir, un orden topológico), con el cual evaluar la expresión dentro de la sumatoria.

5.1. Solución Naive

La manera más sencilla de obtener las clases y sus representantes consiste en calcular todos los órdenes topológicos del bosque utilizando algún algoritmo de generación de los mismos [**?**]. Luego, iterar sobre los órdenes topológicos y asignarles a una clase de equivalencia en base a cuáles nodos no relacionados

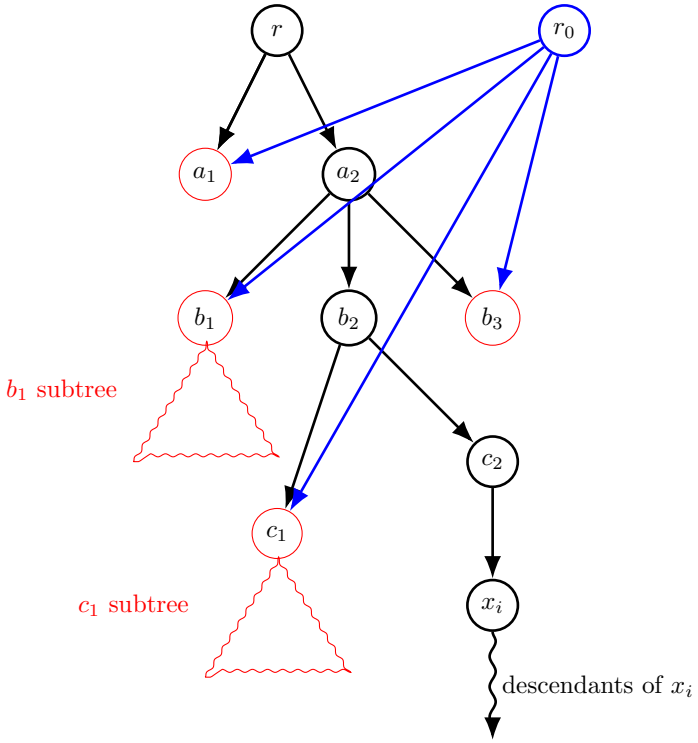


Figura 13: Estrategia utilizada para calcular las clases de equivalencia de los nodos no relacionados, no de todo el DAG.

están antes de x_i. Una vez hecho esto, tendremos nuestras clases, un representante para cada una de ellas y sus tamaños. El problema de este algoritmo es que necesitamos calcular explícitamente cada orden topológico de G, lo cual puede ser $O(n!)$ en el peor caso, ya que esa es la cota para todos los ordenes topológicos posibles.

5.2. Algoritmo recursivo

En este trabajo proponemos un algoritmo recursivo para calcular las clases de equivalencia sin calcular explícitamente todos los órdenes topológicos. El algoritmo para encontrar este conjunto de clases se divide en dos partes: en la primera obtenemos las clases de equivalencia para los árboles no relacionados (es decir, aquellos subárboles que contienen nodos que no son ni descendientes ni ancestros de x_i), y en la segunda fusionamos estas clases con los ancestros y descendientes. En la Figura 14 podemos ver cómo quedan etiquetados los distintos nodos:

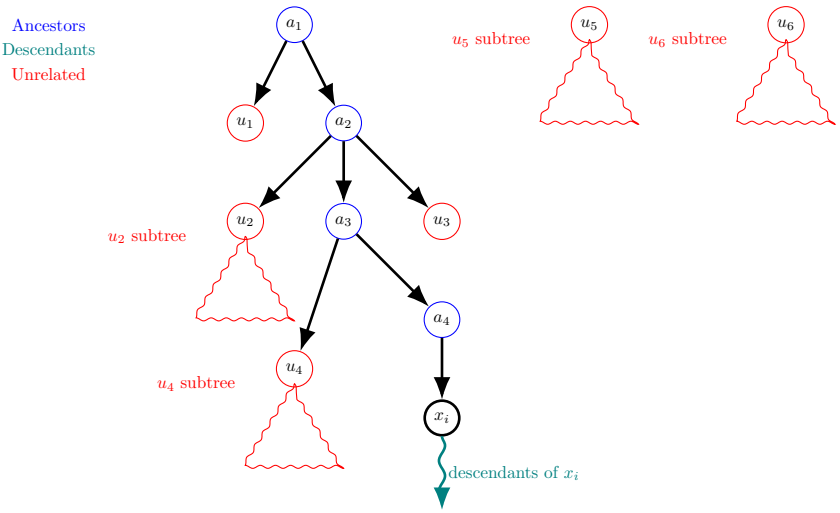


Figura 14: Ejemplo de un grafo causal G, con sus respectivas etiquetas en base a x_i

5.2.1. Clases de equivalencia para árboles unrelated

Sea UR (unrelated roots) el conjunto de nodos que son raíces de un árbol no relacionado. Formalmente, $ur \in UR$ sii ur es una raíz y un nodo no relacionado, o el padre de ur es un ancestro de x_i y ur no es un ancestro de x_i. El algoritmo se ejecutará sobre cada una de estas raíces para obtener las clases de equivalencia de estos subárboles. Cada clase de equivalencia puede representarse con un conjunto $\mathsf{Rep}([\pi]_R)$, como vimos en la Definición 4.1.

Nuestro algoritmo devolverá un conjunto de tuplas con el formato $(\mathsf{Rep}([\pi]_R), leftTopos, rightTopos)$, donde $leftTopos$ es el número de órdenes topológicos que podemos generar con los nodos no relacionados previos a x_i, y $rightTopos$ lo mismo, pero con los que están después. El tamaño de la clase $[\pi]_R$ se puede calcular mediante esta tupla como $leftTopos * rightTopos$.

Para la fórmula a continuación, dado un nodo $n \in V$ notamos como n_i al $i-$ésimo hijo de n, y como $|n|$ a su número de hijos. La Ecuación[11] 6 calcula todas las clases de equivalencia posibles en un árbol no relacionado, dada la raíz del mismo. Aplicándola sobre cada nodo $ur \in UR$, obtendremos el conjunto de clases de equivalencia para cada subárbol no relacionado con x_i. La función $UnrEC(n)$ consiste en obtener todas las clases de equivalencias de los hijos del nodo n, para luego unificar cada combinación posible de las mismas. El caso base es cuando n es una hoja. Ahí solo hay dos opciones: n puede estar a la derecha o a la izquierda, y va a tener un solo orden topológico.

$$UnrEC(n) = \begin{cases} \{(\{n_l\}, 1, 1), (\{n_r\}, 1, 1)\} & \text{si } n \text{ es una hoja} \\ \left(\bigcup_{\forall mix \in UnrEC(n_1) \times \cdots \times UnrEC(n_{|n|})} union(mix, n_{left}) \right) & \text{cc} \\ \cup\, union(right, n_{right}) & \end{cases} \tag{6}$$

La función *union* nos sirve para combinar las distintas clases de equivalencia de cada subárbol para una explicación más detallada de la misma, pueden ir al apéndice a la sección 10.1.1. Para tener una intuición respecto a la correctitud de la función, se puede observar que hay una correspondencia directa entre las clases de equivalencia de un nodo y las de sus hijos. Dado un nodo n, cada clase de equivalencia que lo incluye puede descomponerse de manera única en una combinación de clases de equivalencia de sus hijos. Análogamente, si conocemos todas las clases de equivalencia posibles de los hijos de n, podemos combinarlas para reconstruir todas las clases de equivalencia de n. Esta biyección es la que permite realizar el proceso recursivo definido en la ecuación 6.

Por fuera de la unión, tenemos una clase de equivalencia más, la cual tiene al nodo n a la derecha de x_i. Utilizamos *right* pues es la única unión de clases de equivalencia en la cual todos sus nodos aparecen después de x_i, y si n está a la derecha, entonces todos sus descendientes deben estar a la derecha también. Observar que *right* pertenece al mismo producto cartesiano generado en la unión.

[11] Pueden encontrar este algoritmo en `\pasantia-BICC\asvFormula\classesSizes\recursiveFormula.py`

Optimizaciones Más adelante en este algoritmo vamos a tener que combinar cada uno de nuestros árboles no relacionados con los ancestros. Esa parte del proceso es la más costosa en tiempo. Por lo tanto, nuestro objetivo es terminar con la menor cantidad de subárboles para combinar. En base a esta idea, encontramos esta optimización. Imaginemos que tenemos los resultados para $UnrEC(ur_1)$ y $UnrEC(ur_2)$. Si ur_1 y ur_2 tienen el mismo padre o ambos son raíces, entonces los vamos a unificar. El procedimiento es análogo a agregar un nodo virtual v en la Figura 14, que es el padre de u_5 y u_6, y luego ejecutamos $UnrEC$ desde este nodo, utilizando únicamente a *union* (ya que este nodo no existe y nunca podría estar a la derecha de x_i). De esta forma solo vamos a tener como máximo un árbol no relacionado por cada ancestro de x_i. Otra forma más visual de interpretar esta optimización es que en vez de correr nuestro algoritmo sobre cada ur, lo vamos a ejecutar sobre cada ancestro a. Teniendo el cuidado de no recorrer el eje que conecta al ancestro a a x_i.

5.2.2. Fusión de clases de unrelated trees con ancestros y descendientes

Ahora nuestro objetivo es combinar las clases obtenidas previamente con los ancestros y descendientes de x_i. ¿Por qué no podemos usar la fórmula anterior y simplemente combinarlas como hacíamos antes? Imaginemos que lo hacemos y usamos directamente *union* con los ancestros. La tupla que los representa sería $(\{a_l | a \in A\}, 1, 1)$, $A = ancestros$. Esto se debe a que solo tienen un orden posible, y todos ellos deben aparecer antes de x_i. Ahora bien, si usáramos *union*, estaríamos haciendo el cálculo $\binom{|A|+|U|}{|A|}$, lo que significa que podríamos insertar los elementos de A entre cualquier elemento de U (nodos no relacionados) en el orden topológico, ¡pero eso es incorrecto! Porque en este escenario, hay dependencias entre los nodos. En la Figura 14 no podemos poner u_1 antes de a_1 o ninguno de los nodos en el subárbol de u_2 antes de a_2. Esto significa que necesitamos otra función para calcular los órdenes posibles de los nodos que aparecen antes de x_i y los ancestros. Vamos a llamar a esta función $leftOrders$.

Ahora la pregunta es, ¿qué hacemos con los descendientes? Aquí no estamos restringidos como lo estábamos antes con los ancestros, por lo que podemos usar una fórmula similar a la del algoritmo 4.3, porque no hay dependencias entre los nodos de D y UR. Teniendo esto en cuenta, finalmente podemos definir nuestra fórmula[12] 7. Sea un *dtree* G, un nodo $x_i \in V(G)$, ur_i la raíz del $i-$ésimo árbol no relacionado, A y D los ancestros y descendientes, respectivamente, de x_i en G. Utilizando las funciones $eqCl$ y $eqClassTopos$, las cuales nos devuelven la representación de la clase $\mathsf{Rep}([\pi]_R)$ y el tamaño de la clase, respectivamente. Con estas funciones auxiliares, las cuales están definidas en el apéndice en la sección 10.1.3, la fórmula para calcular todas las clases de equivalencia y sus tamaños en G es:

$$eqClassSizes(G, x_i) = \bigcup_{\forall mix \in UnrEC(ur_1) \times \cdots \times UnrEC(ur_{|UR|})} (eqCl(A, D, mix), eqClassTopos(A, D, mix)) \tag{7}$$

5.2.3. Combinando los ancestros y los nodos no relacionados

Estas son las variables que vamos a tomar en cuenta para calcular las combinaciones: los *ancestros*, el valor de $L(eqCl)$ (la cantidad de nodos a la izquierda en la clase de equivalencia) para cada $eqCl$ de cada unrelated tree, y la raíz de cada árbol no relacionado. Lo que queremos hacer es definir el número de órdenes topológicos que podemos generar combinando los nodos a la izquierda de cada unrelated tree con los ancestros. Veamos un ejemplo con el grafo de la Figura 14. ¿Qué órdenes topológicos podemos generar con los ancestros posicionados como vemos en la Figura 15?

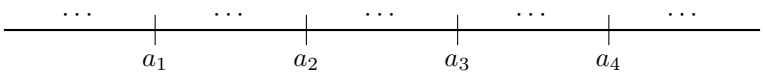


Figura 15: Posible orden topológico con los ancestros ya colocados

Ahora la pregunta es, ¿dónde podemos colocar los elementos de cada unrelated tree? Antes de a_1, solo podemos colocar nodos que estén en los subárboles de u_5 o u_6. Antes de a_2, podemos colocar los mismos elementos, más u_1. En general, antes de un nodo a_i podremos colocar todos los nodos que estén incluidos en un *unrelated tree* que tenga como raíz ur, tal que ur sea una raíz, o a_j sea el padre de u_r

[12] Pueden encontrar este algoritmo en `\pasantia-BICC\asvFormula\classesSizes\recursiveFormula.py`

con $j < i$. Ahora queremos convertir esto en una fórmula recursiva, ya que habrá una superposición de problemas utilizamos programación dinámica al implementar el algoritmo. La fórmula más detallada la pueden encontrar en el apéndice en la sección 10.1.2. Este es el algoritmo para contabilizar las combinaciones posibles entre los nodos no relacionados y los ancestros:

Algorithm 2 leftOrders(A, *actual ancestor*, *nodes to place*, *position*)

1. Definimos donde colocar *actual ancestor* en base a *position* y a cuántos nodos tenemos disponibles en *nodes to place*, generando *new position*.
2. Luego seleccionamos cuántos nodos de cada unrelated tree vamos a usar para llenar todas las posiciones entre *position* y *new position*, generando *new nodes*.
3. Eliminamos los *new nodes* de los *nodes to place*, puesto que ya los colocamos, actualizando nuestros nodos disponibles.
4. Realizamos el llamado recursivo actualizando la posición, nuestros nodos disponibles y nuestro ancestro actual.

5.3. Complejidad del algoritmo

Sea n el número de nodos del digrafo causal G, r la raíz de G y $equivalenceClasses$ el conjunto de clases de equivalencia en G para el feature x_i.

5.3.1. Complejidad temporal de UnrEC

Para calcular la complejidad de UnrEC, definida en la Fórmula 6, vamos a calcular el costo de cada nodo del árbol de llamadas recursivas y el tamaño de este árbol. La complejidad de *union* es $O(n)$, su justificación se encuentra en el apéndice en la sección 10.1.1. Luego necesitamos acotar el número de *mix* que se generará en cada llamada, pero debido a que estamos generando las clases de equivalencia, sabemos que estará acotado [13] por $|equivalenceClasses|$. Por lo que la cota para la complejidad de calcular esta función en cada estado es $O(n * |equivalenceClasses|)$. Luego esta función se ejecuta una sola vez por cada nodo, por lo que se llama $O(n)$ veces, con lo cual la complejidad temporal total es $O(n^2 * |equivalenceClasses|)$.

5.3.2. Complejidad temporal total

La complejidad de eqClassSizes es $O(UnrEC) + |equivalenceClasses| * O(eqClassTopos)$, ya que primero necesitamos ejecutar $UnrEC$ para obtener las clases de equivalencia que vamos a fusionar. Luego, para cada clase de equivalencia que creemos necesitamos calcular su tamaño y sus elementos. Para $eqClassTopos$ sabemos que su complejidad temporal es de $O(n^5 * |equivalenceClasses|^2)$, en la sección 10.1.3 del apéndice se encuentra la justificación de esta complejidad. Por lo tanto, la complejidad total del algoritmo completo será $O(n^2 * |equivalenceClasses|) + O(n^5 * |equivalenceClasses|^3) = O(n^5 * |equivalenceClasses|^3)$.

[13] En el apéndice, en la sección 10.1.2, se puede encontrar una justificación más detallada de esta cota

6. Sampleo de toposorts en polytrees

En esta sección investigamos un enfoque distinto al de las secciones previas, en las cuales buscamos calcular todas las clases de equivalencia y sus representantes, para poder luego computar el ASV de forma exacta. En cambio, ahora presentaremos un algoritmo probabilístico aproximado basado en la idea de samplear órdenes topológicos del DAG causal G. Este nuevo algoritmo, a través de un mecanismo que pueda generar los órdenes topológicos de forma uniforme, busca estimar el valor de ASV con una buena precisión, en base a la cantidad de muestras tomadas. De hecho, como veremos, la cantidad de sampleos necesarios crece lentamente con la precisión deseada, por lo que el método resulta eficiente. La formalización de esta idea se detalla en la sección 10.2.3 del apéndice.

El objetivo inicial consiste en devolver un orden topológico aleatorio para un DAG cualquiera. Nuestro primer acercamiento fue utilizar un algoritmo que genera todos los órdenes topológicos, en nuestro caso el algoritmo de Knuth [**?**], para luego ir devolviendo los órdenes mientras los generábamos. El problema que tiene este enfoque es que este proceso no es realmente aleatorio por varios motivos. Primero, siempre vamos a devolver los mismos y además en el mismo orden, pues el algoritmo de Knuth es determinístico. Además, no vamos a estar teniendo en cuenta la distribución de los mismos si el 99 % de los órdenes comienzan con los mismos nodos, entonces nos gustaría que eso se vea reflejado en nuestro sampleo, y en el caso de utilizar un algoritmo como el de Knuth, nos podría pasar que los órdenes que devolvamos sean muy poco probables.

Otra alternativa es generar todos los órdenes y samplear uniformemente de los mismos. La dificultad radica en que generar todos los órdenes es demasiado costoso ($O(n!)$), y en el caso de generarlos, sería mejor agruparlos en las clases de equivalencias correspondientes y utilizar el algoritmo exacto, en vez de samplear de los mismos. Por lo tanto, esta idea tampoco resulta muy útil en la práctica.

Así es como llegamos a nuestro algoritmo de sampleo, el cual depende de poder contar la cantidad de órdenes de un DAG para poder samplear correctamente.

6.1. Algoritmo de sampleo

Para dar una intuición de este algoritmo vamos a utilizar la Figura 16. Este es un grafo con 600 órdenes topológicos, 100 de estos comienzan con $source_1$, 300 con $source_2$ y 200 con $source_3$. Definimos a S como el conjunto de nuestros nodos fuente (source nodes). Teniendo en cuenta lo mencionado previamente, querríamos que el orden que sampleemos tenga una probabilidad mayor de comenzar con $source_2$ que con los otros dos nodos fuente, ya que hay una mayor cantidad de órdenes que comienzan con ese nodo. Nuestro algoritmo utiliza esta idea para obtener un orden topológico. Primero, según la cantidad de órdenes topológicos que comienzan con cada uno, define una distribución para cada nodo, que asigna una probabilidad proporcional a la cantidad de órdenes topológicos que comienzan con ese nodo. Luego vamos a elegir nuestro siguiente nodo del orden en base a esa distribución, y por último vamos a remover del grafo nuestro nodo elegido, para seguir generando el orden recursivamente a partir de los nodos restantes.

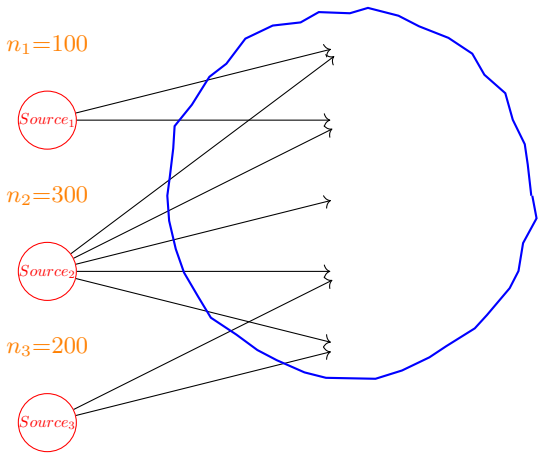


Figura 16: Posible comienzo del algoritmo de sampleo, con los candidatos a ser el primer nodo del orden en rojo y el resto del grafo en azul. Cada node fuente (source) tiene sus respectivas cantidades de órdenes topológicos en los que está primero.

Algorithm 3 SampleoTopoSort(D)

1. **Calculamos una probabilidad** p para cada uno de los nodos fuente del DAG.
   a) Para cada $s \in S$ lo removemos del DAG D, y contamos la cantidad de órdenes topológicos en $D - \{s\}$ ($toposorts_s$), este valor es la cantidad de órdenes que comienzan con s.
   b) Luego a cada $s \in S$ le asignamos una probabilidad $p(s) = \frac{toposorts_s}{\#topos(D)}$.
2. **Sampleamos** sobre S utilizando p para obtener nuestro primer nodo *start*.
3. **Eliminamos** a *start* de D y llamamos al algoritmo recursivamente con SampleoTopoSort($D - \{start\}$), guardando el resultado en *orden*.
4. **Devolvemos** $start + orden$ como el orden topológico sampleado.

En la implementación de este algoritmo[14] también utilizamos una caché para el número de órdenes topológicos de cada subgrafo del DAG, de esta forma nos ahorramos calcular más de una vez el conteo de los órdenes para un mismo subgrafo.

Para nuestro caso de uso vamos a necesitar más de un orden, ya que necesitamos samplear varios órdenes. Por lo que ahora nuestra función va a ser `SampleoTopoSorts`(D, k), la cual recibe k además de D, y devuelve k órdenes topológicos aleatorios del DAG D. La implementación naive de esta función sería llamar k veces a `SampleoTopoSorts`. Pero si hacemos esto vamos a estar repitiendo varias veces los mismos pasos, cuando los podríamos hacer en simultáneo. Vamos a calcular k veces la probabilidad p para los nodos fuente de D, y eliminaremos k veces cada nodo elegido de D. En cambio, una alternativa mejor consiste en calcular una vez p y obtener los k nodos necesarios para cada iteración. Por lo tanto, hay que realizar dos modificaciones a nuestro Algoritmo 3 para no realizar estas operaciones innecesariamente. En el paso 2, vamos a samplear k nodos de S (puede haber nodos repetidos). Esto nos va a devolver un número k_s para cada $s \in S$ ($\sum_{s \in S} k_s = k$), que es la cantidad de órdenes que van a comenzar con ese nodo. Luego vamos a llamar a `SampleoTopoSorts`($D - s$, k_s) para cada s con $k_s > 0$, y agregar s al comienzo de cada uno de los k_s órdenes obtenidos.

6.2. Número de órdenes topológicos de un polytree

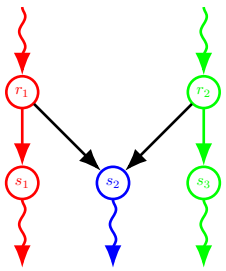


Figura 17: Ejemplo de polytree para el cual no funciona la Fórmula 4.3, la cual cuenta los órdenes topológicos de un *dtree*. Los descendientes en común y disjuntos de cada raíz están marcados de un color distinto.

Para poder utilizar este algoritmo de sampleo necesitamos poder contar el número de órdenes topológicos de nuestro DAG. Actualmente, ya podemos contar los órdenes de un *dtree*, pero nuestro objetivo era poder tratar con una familia de grafos más amplia, por lo que vamos a volver al ejemplo para el cual la Fórmula 4.3 no podía contabilizar correctamente el número de órdenes topológicos. En la Figura 17 no podemos usar la misma estrategia que para el árbol, puesto que hay un solapamiento entre los subárboles de r_1 y r_2, todos los descendientes de s_2. A partir de esto, el caso que queremos resolver es cuándo un nodo tiene dos o más padres. Para resolver este problema, comenzamos con un enfoque similar al realizado en el Algoritmo 4.3 para *dtrees*, pero nos encontramos con un caso para el cual no funcionaba. Luego terminamos cambiando el enfoque y llegamos a un nuevo algoritmo, el cual nos permitió calcular todos los órdenes topológicos de cualquier polytree.

[14] Pueden encontrar este algoritmo en `\pasantia-BICC\asvFormula\topoSorts\randomTopoSortsGeneration.py`

6.2.1. Idea inicial

Nuestra idea para el resolver el caso de la Figura 17 consiste en calcular los resultados de los subárboles de r_1, s_2 y r_2 por separado, para luego unificar sus resultados. En la Figura 18 tenemos 3 órdenes posibles. Nuestro objetivo es combinarlos en un orden π. Las únicas restricciones que tenemos que tener en cuenta al combinarlos es que $\pi(s_2) < \pi(r_1) \wedge \pi(s_2) < \pi(r_2)$, ya que s_2 es un descendiente de ambos nodos.

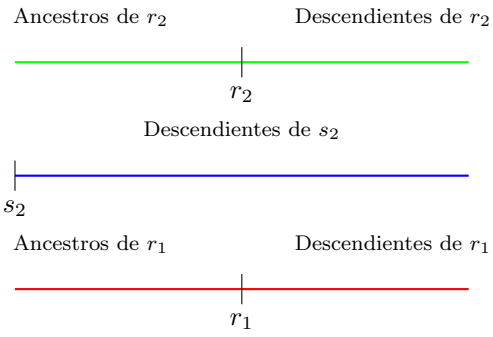


Figura 18: Estos serían tres órdenes topológicos posibles, uno para cada subárbol de la Figura 17. s_2 es la raíz de su subárbol, por lo que siempre va a estar primero.

Sabemos que estos 3 conjuntos de nodos no van a tener otras aristas que los conecten, ya que en ese caso no sería un polytree, pues se generaría un ciclo en el grafo subyacente. Por esto podemos combinar sus órdenes sabiendo que no va a haber solapamiento. De esta forma, nuestro algoritmo va a consistir en remover las aristas de los nodos que tienen múltiples padres, calcular el resultado para cada uno de sus subárboles y luego combinar estos resultados para obtener todos los órdenes.

Nuestro primer intento se basó en utilizar el multinomial para combinar los órdenes, pero el problema de esta fórmula es que no tiene en cuenta la restricción pedida, que r_1 y r_2 aparezcan previamente a s_2. Otra opción sería combinar todos los órdenes de r_1 y r_2 para luego insertar los órdenes de s_2. Pero nos volvemos a encontrar con el mismo problema si combinamos los órdenes de r_1 y r_2 utilizando el multinomial, nos vamos a perder la información de dónde se encuentran estos dos nodos, y por lo tanto a partir de qué posición podemos colocar a s_2. Así es como llegamos a la conclusión de que necesitábamos calcular no solo la cantidad de órdenes topológicos y el tamaño de cada subárbol, sino que también debíamos guardarnos la posición de los padres r_i en cada uno de esos órdenes.

Para eso hicimos la función `positionInTopsorts`(D, n)(o *pit*), que dado un *dtree* D y un nodo n, retorna un conjunto *toposPositions*, el cual contiene duplas de la forma $(position, toposorts) \in toposPositions$, siendo *toposorts* la cantidad de órdenes topológicos del *dtree* D, en los cuales el nodo n se encuentra en la posición *position*. Para nuestro ejemplo tenemos que calcular *positionInToposorts* para los tres conjuntos de nodos marcados.

Vamos a definir a $i_\cap$ como nuestro nodo con más de un padre, en nuestro ejemplo sería s_2. Ahora queremos ver cómo combinar los resultados obtenidos al evaluar *positionInToposorts* sobre cada uno de los padres e $i_\cap$[15]. Como los padres no tienen ninguna restricción entre sí, pueden aparecer en cualquier orden. Luego, para cada uno de esos ordenamientos posibles, buscamos cuántos órdenes topológicos hay que respeten ese ordenamiento. Para calcular eso vamos a utilizar una función similar a la vista en el Algoritmo 2, llamaremos a esta función modificada $allPosibleOrders(before, after, parents, topos)$[16] (o *apo*). Esta función nos va a devolver todos los órdenes topológicos dado un ordenamiento de los padres y una lista de duplas $(pos, topo)$ (el output de *positionInTopsorts*) para cada padre. A diferencia de #LO, los nodos que vamos a colocar no son los ancestros de x_i, sino los padres de $i_\cap$. Otra diferencia es que además de tener los nodos a la izquierda de los padres, también vamos a tener los nodos a la derecha para colocar. Los nodos a la izquierda de cada padre los obtenemos con su posición y los nodos a la derecha utilizando su posición y el tamaño del subárbol del padre.

Así llegamos a la función que nos va a permitir calcular los órdenes topológicos aunque haya más de un padre. Lo que vamos a hacer es iterar por todos los ordenamientos de los padres y luego iterar por todas las combinaciones posibles del output de *positionInToposorts*. Los parámetros de la función son:

- $i_\cap$: el nodo intersección.

[15] Cuando evaluamos $positionInToposorts(D - \{(n, i_\cap)\}, p)$ con $p \in parents(i_\cap)$, removemos la arista que conecta p a $i_\cap$ puesto que si no estaríamos calculando dos veces lo mismo, ya que vamos a correr $positionInToposorts(D, i_\cap)$.

[16] Pueden encontrar este algoritmo en `\pasantia-BICC\asvFormula\topoSorts\topoSortsCalc.py`

- ps : los padres de $i_\cap$.
- $poly$: el polytree que contiene los subárboles s_p para cada $p \in ps$.

$$ordersFromIntersection(i_\cap, ps, poly) = \sum_{p \in perm(ps)} \sum_{\substack{topos \in \\ pit(p[1], s_{p[1]}) \times \cdots \times pit(p[|p|], s_{p[|p|]}) \times pit(i_\cap, s_{i_\cap})}} apo(topos) \tag{8}$$

Así llegamos a la versión inicial de nuestro algoritmo. Primero recorremos el polytree en `intersectionNodes` para obtener los distintos nodos con más de un padre. Luego para cada intersección utilizamos `ordersFromIntersection` para obtener todos los órdenes. Por último, combinamos los resultados obtenidos para cada polytree al igual que con los *dtrees*, ya que no hay aristas entre los distintos polytree por lo que no hay restricciones al combinarlos. La versión inicial del algoritmo[17]:

Algorithm 4 Versión inicial - Número de órdenes topológicos en polytrees

```
1:  function allPolyTopoSorts(polyTree)
2:      intersectionNodes ← intersectionNodes(polyTree)
3:      topos ← {}
4:      sizes ← {}
5:      for inter ∈ intersectionNodes do
6:          topos ← topos ∪ ordersFromIntersection(inter, parents(inter), polytree)
7:          sizes ← sizes ∪ totalSize(polytree, inter)
8:      end for
9:      totalToposorts ← multinomialCoefficients(treeSizes)
10:     totalToposorts ← totalToposorts · ∏_{t∈topos} t
11:     return totalToposorts
12: end function
```

Ahora podemos resolver el caso de la Figura 17, y más general aún, podemos resolver cualquier DAG que sea un polyforest que tenga sólo un nodo con múltiples padres en cada uno de sus polytrees. ¿Qué ocurre si tenemos más de una intersección como en la Figura 19?

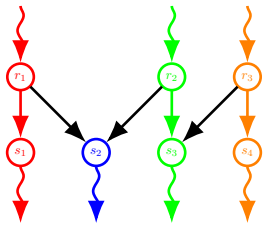


Figura 19: Ejemplo de polytree para el cual no funciona el algoritmo 4 para contar los órdenes topológicos en polyforests. Cada subárbol que vamos a tener en cuenta está marcado de un color distinto.

El problema yace en que para poder utilizar *ordersFromIntersection*, necesitamos poder aplicar `positionInTopsorts` sobre cada uno de los subárboles y que estos sean *dtrees*. Pero sin importar si comenzamos por los padres de cualquiera de nuestras intersecciones (s_2 y s_4), vamos a tener un subárbol el cual no es un *dtree*. Al no poder solucionar este caso a través de este primer enfoque, repensamos el algoritmo desde cero, buscando un enfoque distinto. A continuación veamos el algoritmo que surgió de este cambio, el cual terminó siendo el algoritmo para el caso general.

6.2.2. Algoritmo final

La idea surge de ver cómo calcular los órdenes desde un nodo, teniendo en cuenta que ya tenemos calculados los de todos los vecinos. Sabiendo lo que necesitamos para combinarlos, la posición

[17] Pueden encontrar este algoritmo en `\pasantia-BICC\asvFormula\topoSorts\topoSortsCalc_basic.py`

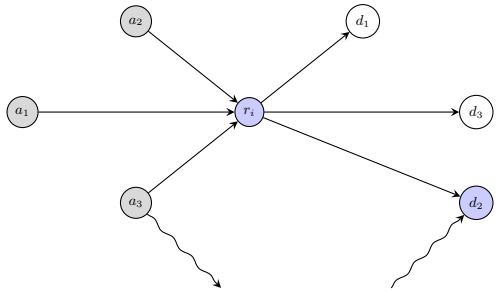


Figura 20: DFS enraizado en r_i: los nodos grises ya fueron visitados; los azules está en proceso y los blancos no fueron procesados todavía.

de los vecinos, los órdenes y el tamaño de los subárboles, solo queda ver cómo unirlos. El algoritmo consiste en realizar un DFS sobre el grafo subyacente del polytree para obtener sus órdenes topológicos. La función `polytreeToposorts`($node$, D) (o $polyTopo$) [18] va a retornar el mismo resultado que `positionInTopsorts`, dado un polytree D y un nodo $node$ devuelve el conjunto $toposPositions$ para ese nodo (solo que esta función no necesita que D sea un $dtree$). Por ejemplo, en la Figura 20 comenzamos por r_i revisando sus vecinos. Por ejemplo, al llegar a d_3, vamos a obtener sus vecinos *no visitados* para calcular sus órdenes, al no tener ninguno vamos a devolver el conjunto $\{(0,1)\}$, ya que solo tiene un orden y se encuentra primero en ese orden. Este es nuestro caso base. Ahora queda ver el caso en el cual tenemos vecinos que no fueron visitados. Primero necesitamos obtener los resultados de todos nuestros vecinos, por lo que vamos a hacer la llamada recursiva en cada uno de ellos. Luego, para combinar los resultados de los distintos vecinos no visitados, vamos a utilizar $combineNodesOrder(n, ord, poly)$, la cual recibe un orden de los vecinos no visitados de n y calcula el número de órdenes topológicos . Esta función utiliza los resultados de haber corrido $polytreeToposorts$ sobre cada uno de los vecinos. Para calcular todos los órdenes vamos a utilizar $allNVNeighbourOrders(D, node)$, que nos devuelve todas las permutaciones posibles de los vecinos *no visitados* de $node$ (con $node$ incluido) que sean un orden topológico válido [19].

Una vez que tenemos nuestro ordenamiento de los vecinos no visitados y sus resultados, sólo queda combinarlos. Para eso vamos a iterar sobre el producto cartesiano de los resultados obtenidos, calculando para cada tupla todos sus órdenes y posiciones, utilizando una nueva función. Su input va a ser: el nodo n, un orden $(p_i, \ldots, n, \ldots, h_i)$, siendo p_i un padre y h_j un hijo de n y una tupla perteneciente al producto cartesiano $((pos_{p_i}, top_{p_i}), \ldots, (pos_{h_j}, top_{h_j}))$. La función $posAndOrd$ también nos va a devolver un conjunto como el de $toposPositions$, utilizando $allPosibleOrders$ para calcular los ordenamientos posibles y haciendo cálculos similares a los vistos en esta sección para obtener el conjunto de tuplas $(position, topos)$ para n. Por último, en la unión de $combineNodesOrder$ se van unificando las tuplas $(p1, t1), (p2, t2), \ldots, (p_i, t_i)$ con $p1 = p2 = \cdots = p_i$ en $(p1, t1 + t2 + \cdots + t_i)$, por lo que no va a tener dos tuplas con la misma posición. Así llegamos a la fórmula:

$$combineNodesOrder(n, o, poly) = \bigcup_{\substack{topos \in \\ polyTopo(o[1], poly) \times \cdots \times polyTopo(o[k], poly)}} posAndOrd(topos, o, n) \quad (9)$$

Una vez que definimos $combineNodesOrder$, nuestro algoritmo final consiste en hacer DFS, combinando los resultados de los vecinos no visitados. En `unifyNeighboursResults` hacemos lo mismo que en $combineNodesOrder$, unificando las tuplas con la misma posición. De este modo, presentamos la versión final del Algoritmo 5:

6.3. Complejidad del algoritmo

Vamos a analizar la complejidad del Algoritmo 5. Para eso veamos cuál es el costo de procesar cada uno de los nodos y cuántas veces vamos a llamar a esta función por cada nodo.

[18] Pueden encontrar este algoritmo en `\pasantia-BICC\asvFormula\topoSorts\topoSortsCalc.py`

[19] Para que sea un orden topológico válido los padres de $node$ tienen que estar ubicados antes que $node$ y $node$ antes de sus hijos. A diferencia del enfoque anterior, ahora vamos a estar revisando los padres y los hijos a la vez, por lo que hay que tener en cuenta esta restricción para las permutaciones.

Algorithm 5 Final - Número de órdenes topológicos para polytrees

```
1: function polytreeTopsorts(D, node)
2:     topos ← {}
3:     if |unvisitedNeighbors(node, D)| = 0 then
4:         return {(0, 1)}
5:     end if
6:     for order ∈ allNVNeighbourOrders(D, node) do
7:         topos ← topos ∪ combineNodesOrder(node, order, polytree)
8:     end for
9:     toposPositions ← unifyNeighboursResults(topos)
10:    return toposPositions
11: end function
```

Vamos a evaluar cuánto cuesta el llamado en base a *node* y D. Obtener $allNVNeighbourOrders(D, node)$ va a costar $O(d_{out}(v)! \cdot d_{in}(v)!)$, pues ese es el número de todas las permutaciones posibles. Sean $v_1, \cdots, v_k$ los k vecinos de *node* y $s_1, \cdots, s_k$ sus subárboles visitados, con s_{max} el subárbol de tamaño máximo. Entonces se cumple que $|positionInTopsorts(v_i, s_i)| \leq |s_i| \leq |s_{max}|$, pues las posiciones en las que puede estar un nodo en un orden topológico están acotadas por la longitud del orden. Así que en $combineNodesOrder$ se realiza la unión de $O(|s_{max}|^k)$ elementos.

Por último tenemos la complejidad de llamar a *posAndOrd*, la cual depende de la complejidad de $allPosibleOrders$. El cálculo de esta complejidad se encuentra en la sección 10.1.4 del apéndice, pero para un grado de vecinos k acotado, se llega a la cota $O(n^{3k-1})$ para su complejidad.

Así nuestra complejidad final nos queda $O(allNVNeighbourOrders) \cdot O(combineNodesOrder) \cdot O(allPosibleOrders)$, siendo k nuestro grado acotado para los vecinos. Así la complejidad total nos queda $O(k! \cdot n^k \cdot n^{3k-1}) = O(k! n^{4k-1})$, para nuestro algoritmo de conteo completo.

7. ASV end to end

7.1. ASV exacto

Recordemos la fórmula introducida en la sección 4 para ASV, utilizando nuestra heurística.

$$\phi_i^{assym}(\nu) = \sum_{\pi \in perm(\mathcal{I})} w(\pi)\left[\nu(\pi_{<i} \cup i) - \nu(\pi_{<i})\right] =$$
$$\frac{1}{|topos(G)|} \sum_{[\pi]_R \in eqCl(G,x_i)} (\nu(\pi_{<i} \cup \{x_i\}) - \nu(\pi_{<i})) \cdot |[\pi]_R|$$

Dado un DAG G, un nodo $x_i \in V(G)$ y una función característica ν. Nuestro algoritmo completo queda así entonces:

- Calculamos $eqClass$ a través de $eqClassSizes(G, x_i)$, el algoritmo que introducimos en la sección 5
- Luego para cada clase de equivalencia vamos a calcular el promedio teniendo en cuenta los features previos a x_i. Para realizar el promedio vamos a utilizar el algoritmo de la sección 3. (En el caso de que el modelo no sea un árbol, podemos aproximarlo utilizando Monte Carlo, pero el algoritmo dejaría de ser exacto)
- Por último sumamos los resultados para obtener él $Shap^{assym}$ para el feature i.

Las modificaciones que introducimos son para mejorar la performance del mismo, sin modificar los resultados obtenidos a diferencia del enfoque aproximado. En la sección a continuación vamos a ver cuál es la mejora respecto al enfoque naive y el tiempo que tarda para distintas redes.

7.2. ASV aproximado

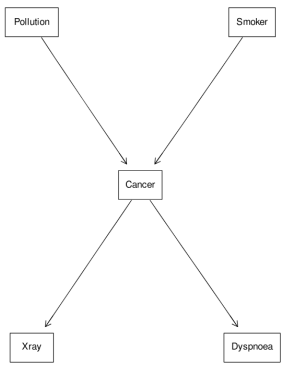
Este algoritmo es igual al anterior. La única diferencia que tiene es respecto a cómo se calculan las clases de equivalencia, $eqClass(G, x_i)$, puesto que ahora las vamos a aproximar. Además, vamos a tener que agregar un parámetro extra, para definir la cantidad de órdenes topológicos que queremos generar. Primero utilizamos el algoritmo 3 para samplear los órdenes topológicos. Esto nos va a permitir calcular él ASV para polytrees y no simplemente *dtrees*. Luego procesamos estos órdenes al igual que en la sección 5.1, para obtener las clases de equivalencia. Una vez obtenidas las clases de equivalencia, repetimos el mismo proceso del algoritmo exacto.

8. Experimentos

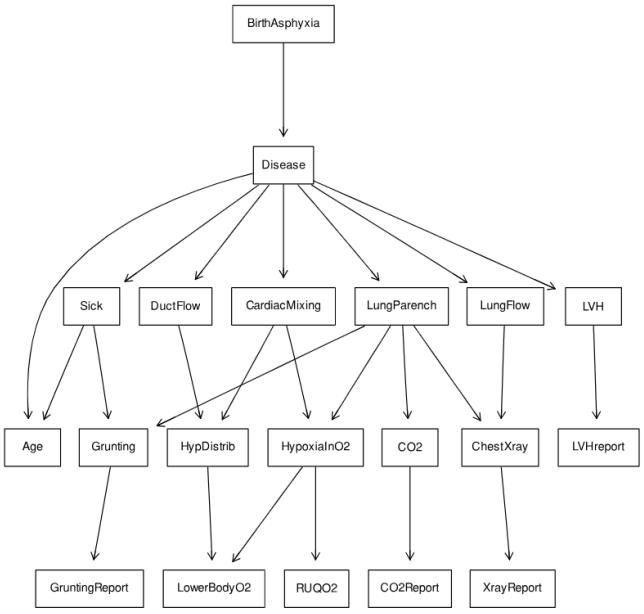
Habiendo definido nuestros algoritmos, lo que nos queda es ver cuál es la ventaja que otorgan los mismos en comparación a la solución naive de cada uno de estos problemas. Esta ventaja puede ser en precisión, tiempo o espacio.

Para los experimentos utilizamos dos redes bayesianas, *Cancer* y *Child*, tomadas del paquete de R bnlearn, podemos ver su estructura en la Figura 21. Utilizamos estas redes, ya que tenían un tamaño manejable y una topología similar a la de un polytree. Además de ser la distribución de nuestros datos, las empleamos como nuestros grafos causales, asumiendo que fueron generadas basándose en la causalidad [20]. En el caso de la red *Child*, al no ser un polytree tuvimos que remover algunas aristas para lograr obtener esa estructura. Analizamos distintas heurísticas para remover ejes de la red convirtiéndola en un polytree. Utilizamos la más sencilla, que consiste en remover los ejes que generan ciclos en el grafo subyacente de la red bayesiana. Una opción más refinada consiste en elegir las aristas a remover, en base a cuáles minimizan la divergencia entre las distribuciones marginalizadas. Esto lo podríamos hacer comparando las distintas distribuciones que nos quedan al remover distintas aristas, utilizando una métrica como la divergencia de Kullback-Leibler, pero no era el objetivo de esta tesis.

Para generar los datasets de entrenamiento y test, sampleamos instancias de ambas redes. Luego entrenamos al DT usando este dataset. A la hora de calcular el ASV, removimos la variable a predecir de nuestra red. Removemos la variable, puesto que si tuviéramos la red bayesiana completa con nuestra variable a predecir en la misma, haríamos la inferencia directamente en la red bayesiana [21]. Las variables que definimos para predecir fueron `Smoker` y `Age` para sus respectivas redes. En el caso de la red *Cancer* no elegimos `Cancer`, puesto que nuestro DAG causal iba a perder todas sus dependencias, por lo cual ASV no iba a poder detectar relaciones significativas. Luego en el caso de *Child* el criterio fue no utilizar una variable que se encuentre muy arriba en el árbol, puesto que no iba a tener muchas variables que la influencien, ni tan abajo que cueste distinguir las variables que la impactan mayormente. Por lo que nuestro input va a ser una red bayesiana N, un feature a predecir p, un dataset $data$, un árbol de decisión DT y una instancia $x \in data$.



(a) Red Bayesiana *Child* para analizar las enfermedades de niños de recién nacidos. Fuente: [?]



(b) Red Bayesiana *Cancer*para determinar la probabilidad de tener cáncer de distintos pacientes. Fuente: [?]

Figura 21: Ejemplos de redes bayesianas utilizadas en los experimentos.

Especificaciones de los experimentos

[20] Esto no necesariamente siempre se cumple, pero entendemos que es una suposición razonable para la mayoría de los casos.

[21] Tampoco entrenaríamos un árbol de decisión, sino que simplemente utilizaríamos a la red para clasificar las instancias

- **Procesador:** Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz
- **Memoria RAM:** 16 GB
- **Sistema operativo:** Ubuntu 22.04 LTS
- **Python:** Versión 3.12
- **Paquetes:** pgmpy (inferencia bayesiana), sklearn, networkx, shap

8.1. Clases de equivalencia vs Órdenes Topológicos

Para este experimento, vamos a comparar la implementación clásica de *ASV* con nuestra idea de utilizar las clases de equivalencia para reducir los términos de la sumatoria. Para eso vamos a comparar la forma original de calcular ASV:

$$\frac{1}{|topos(G)|} \sum_{\pi \in topos(G)} w(\pi) \left[\nu(\pi_{<i} \cup i) - \nu(\pi_{<i})\right]$$

con nuestra heurística:

$$\frac{1}{|topos(G)|} \sum_{[\pi]_R \in eqCl(G,x_i)} (\nu(\pi_{<i} \cup \{x_i\}) - \nu(\pi_{<i})) \cdot |[\pi]_R|$$

Hay dos métricas a tener en cuenta para ver cuál de estas dos estrategias es mejor. Primero, ver cuánto es el tiempo que se tarda en obtener los conjuntos sobre los que efectuar la sumatoria, que son $eqCl(G, x_i)$ y $topos(G)$. Luego comparar el tamaño de cada uno de esos conjuntos, puesto que por cada elemento de ese conjunto vamos a tener que evaluar a ν dos veces. Podría ocurrir que la construcción de las clases de equivalencia resulte computacionalmente costosa, y su cardinalidad no necesariamente presente una reducción significativa en comparación con *topos*. En tales casos, el costo adicional de calcularlas puede superar el beneficio esperado, incrementando el tiempo total de cómputo.

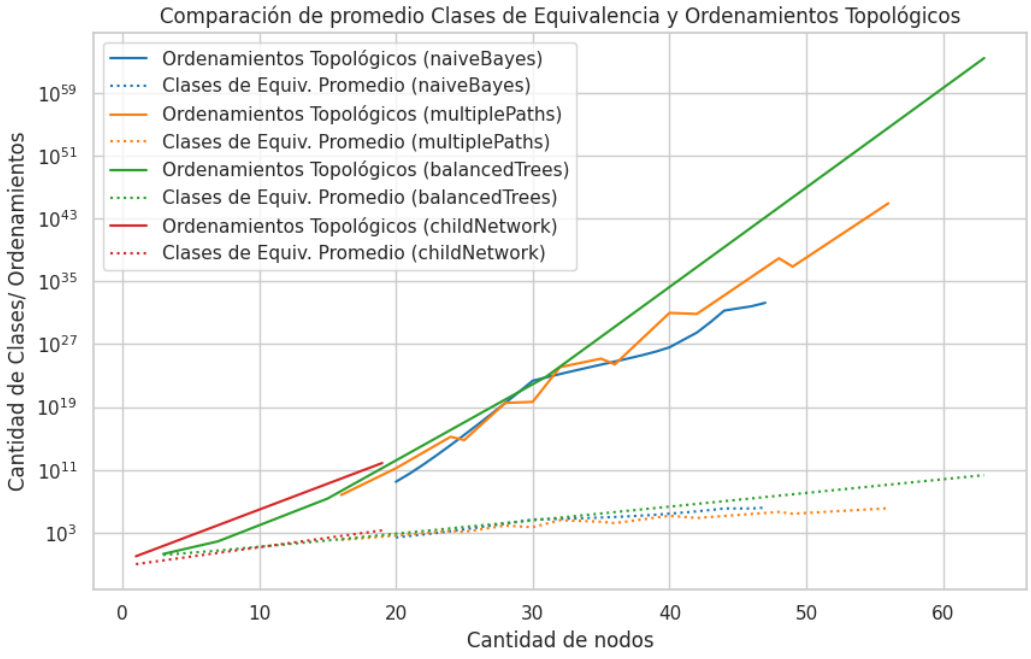


Figura 22: Comparación de clases de equivalencias y órdenes topológicos de distintas clases de grafos. Se utiliza un promedio, puesto que la cantidad de clases de equivalencia depende del nodo que elijamos para calcularlas. [22]

Las distintas clases de grafos mencionados en la Figura 22 son:

[22] La función no es monótona, ya que no se utilizaron todos los grafos posibles para cada una de las clases mencionadas, se realizó una estimación a partir de una muestra significativa de distintos grafos de cada clase.

- Naive Bayes: Una red Naive Bayes con n nodos, que tiene $n/2$ hojas y que tiene un camino de longitud $n/2 - 1$ en una de sus hojas.
- Multiple paths: Un bosque compuesto de múltiples caminos de igual longitud.
- Balanced tree: Un árbol binario perfecto balanceado.
- Child network: La red bayesiana *Child*, sin algunos de sus ejes para ser un polytree.

En la Figura 22, podemos observar como el número de clases de equivalencia crece significativamente más lento que el número de órdenes topológicos. Por ejemplo, en el caso de la red *Child*, la red tiene 7.41×10^{11} órdenes topológicos y 2003 clases de equivalencia en promedio, por lo que si utilizamos las clases disminuimos enormemente la cantidad de llamadas a ν. Luego, para los árboles balanceados, la cantidad de órdenes topológicos es 10^{50} veces mayor, una diferencia muy significativa. A partir de estos ejemplos, queda claro que es una mejora hacer el cálculo sobre las clases de equivalencia. Ahora solo queda ver el costo de calcularlas.

El costo de calcular las clases lo podemos ver en la Figura 23. Para la mayoría de los grafos de ejemplo que utilizamos tarda menos de 10 segundos. Pero para grafos de mayor tamaño, el tiempo que tarda comienza a crecer exponencialmente, al igual que la cantidad de clases de equivalencia. En el caso puntual de la red *Child*tarda menos de 1 segundo en calcular todas sus clases.

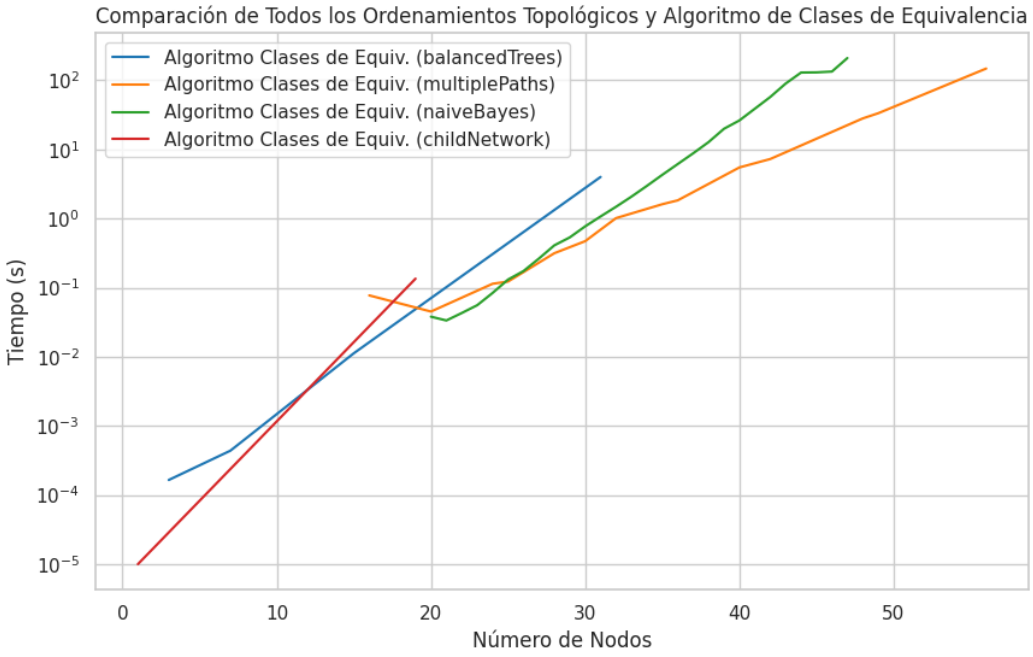


Figura 23: Comparación del tiempo que tarda el algoritmo para calcular las clases de equivalencia

La Figura 24 nos muestra para distintos tipos de grafos cuánto tiempo toma cada algoritmo. En uno se obtienen todas las clases de equivalencia y en el otro todos los órdenes topológicos. A partir de 10^{15} órdenes topológicos el problema de calcularlos todos tardaría días, en cambio, calcular las clases de equivalencia sigue siendo una estrategia eficiente. Esto sucede así, puesto que, como vimos en el lema 1, la cantidad de clases de equivalencia depende de la estructura del grafo, y no de la cantidad de órdenes.

Por ende, a través de este experimento pudimos ver que la cantidad de clases de equivalencias es significativamente menor que la cantidad de órdenes, por lo que se reduce la cantidad de llamadas a ν. Además, en la figura 24, podemos ver cómo el tiempo para calcular las clases de equivalencia crece más lentamente que el tiempo de calcular todos los órdenes. Por lo tanto, podemos concluir que el algoritmo proporciona una mejora del tiempo para calcular todas las clases respecto a la implementación naive.

[23]Para estimar el tiempo que tardaría se calcularon los primeros 1000 órdenes que devuelve el algoritmo exacto. Luego se utilizó ese tiempo para hacer una aproximación del tiempo total, sabiendo la cantidad de órdenes de cada grafo. Esto se realizó por simplicidad, ya que no era viable correr el algoritmo durante tanto tiempo.

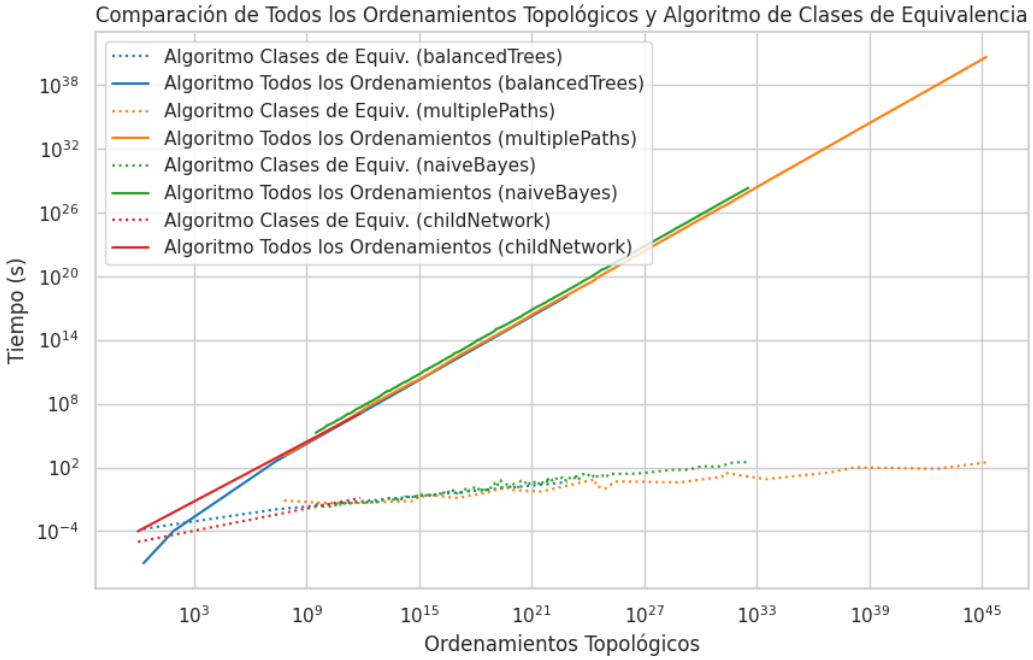


Figura 24: Comparación del tiempo que tarda el algoritmo para calcular las clases de equivalencia y calcular todos los órdenes topológicos, utilizando una implementación de la librería networkx. [23]

8.2. ASV vs SHAP

Este experimento consiste en calcular el valor del ASV y $SHAP$, para cada uno de los features de los modelos utilizados en el experimento de la sección 8.4. Vamos a correr este experimento para 5 seeds distintas, ya que los valores obtenidos pueden depender de la aleatoriedad de los datos y queremos contrastar múltiples resultados. Aun así, vamos a elegir una seed para analizar para cada una de las redes, utilizando como criterio para elegir la seed el modelo con la mejor accuracy, ya que si el modelo no aprendió correctamente los patrones subyacentes de los datos, entonces los valores de ASV y $SHAP$ pueden no correlacionarse con el grafo causal original. Los resultados de todas las corridas pueden verse en el apéndice en las Figuras 32 y 33. Correr el ASV para todos los features de la red *Cancer* tarda 1 segundo, y correrlo para todos los features de la red *Child* tarda 3 minutos.

Para obtener una intuición acerca de ambas redes y las distintas relaciones entre sus nodos nos basamos en [**?**] para la red *Child* y [**?**] para la red *Cancer*.Además, generamos una Tabla 1 para ambas redes, para analizar el impacto de modificar cada una de las variables de la red. Por ejemplo, la probabilidad original de la variable `Smoker` es $P(Smoker = True) = 0.3$ pero si sabemos que el paciente tiene cáncer pasa a ser $P(Smoker = True|Cancer = True) = 0.8255$. Por lo que (como era de esperarse), `Cancer` es una variable significativa a la hora de calcular si un paciente es fumador o no. En este caso, vemos que la probabilidad de que sea fumador se vio modificada en 0.5255 al introducir la evidencia de que tenía `Cancer`. Esta variación de la probabilidad es la que vamos a ver en la columna *Probability Shift* en la Tabla 1.

Comencemos analizando el caso de la red *Cancer*. En la Figura 25 podemos ver que para ASV la única variable significativa es `Cancer`. Esto tiene sentido con lo visto en la Figura 21a, ya que es la única variable conectada directamente con `Smoker`. Pero si nos basáramos en los resultados obtenidos en los Shapley Values, creeríamos que `Xray` y `Dyspnoea` también tienen un impacto significativo en si es fumador o no el paciente. La forma que tenemos para ver cuál de los métodos está detectando correctamente las variables relevantes es la Tabla 1, en la cual podemos ver que la variable que más impacta es `Cancer` y que la `Dyspnoea` tiene un impacto mucho menor. Esta nueva métrica que vamos a utilizar es igual de arbitraria que Shap, pero nos permitió realizar una comparación y un análisis cuantitativo más allá del significado de cada una de las variables. Esto ocurre, ya que ASV tiene en cuenta el grafo causal a la hora de realizar estos cálculos, a diferencia de $SHAP$.

Luego en el caso de la red *Child*, estas son las 5 variables más relevantes para las distintas métricas propuestas:

Variable	Smoker	New smoker probability	Probability Shift
Pollution = Low	True False	0.3 0.7	0.0
Pollution = High	True False	0.3 0.7	0.0
Cancer = True	True False	0.8255 0.1745	0.5255
Cancer = False	True False	0.2938 0.7062	0.0062
Xray = Positive	True False	0.3206 0.6794	0.0206
Xray = Negative	True False	0.2946 0.7054	0.0054
Dyspnoea = True	True False	0.3070 0.6930	0.0070
Dyspnoea = False	True False	0.2969 0.7031	0.0031

Tabla 1: Variación de la probabilidad de ser fumador en base a la nueva evidencia

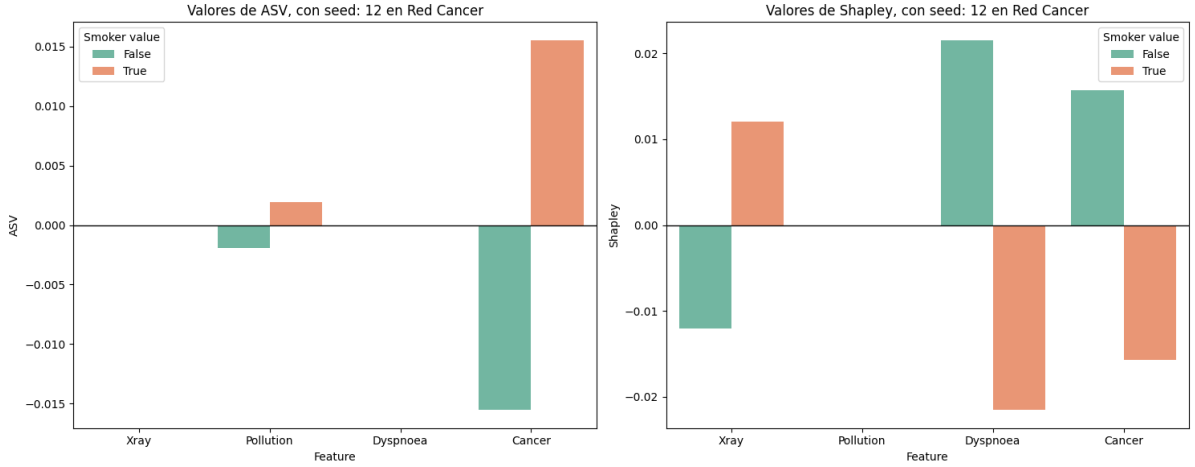


Figura 25: Resultados del ASV y Shapley para el modelo con mayor accuracy de los 5 seeds para la red *Cancer*, con un 81 % de accuracy.

- **Mayor valor de Probability Shift**: Disease, Duct Flow, Sick, Cardiac Mixing, LVH
- **Mayor valor de ASV**: Disease, Duct Flow, Sick, LVH, LVH Report
- **Mayor valor de SHAP**: Disease, ChestXRay, CO2, RUQO2, LVH Report

Estos resultados[24] se basan en la Figura 26. Lo que podemos ver es que la intersección entre los features más relevantes de Probability Shift y ASV, es mayor a la de $SHAP$ con la misma métrica. Ya que aunque ambos logran identificar a los nodos `Disease` y a `LVH/LVH Report` como relevantes, sólo ASV encuentra la relación con `Sick` y `DuctFlow`. Aún así esto podría variar según el modelo, ya que si el modelo no logró identificar correctamente las relaciones entre los datos, los valores de ASV y $SHAP$ tampoco van a correlacionarse con el grafo causal. Analizar estos valores nos puede ayudar a ver si tiene sentido la elección de features más relevantes que está utilizando nuestro algoritmo para realizar sus predicciones.

En base a lo observado en estos dos casos, podemos ver que ASV puede detectar relaciones entre los features que $SHAP$ no logra encontrar. Esto se debe a que utiliza la información del grafo causal, para ver cuáles features priorizar a la hora de realizar estos cálculos.

[24]Los datos completos se pueden encontrar en `\pasantia-BICC\results`, para ver la tabla completa para todas las variables.

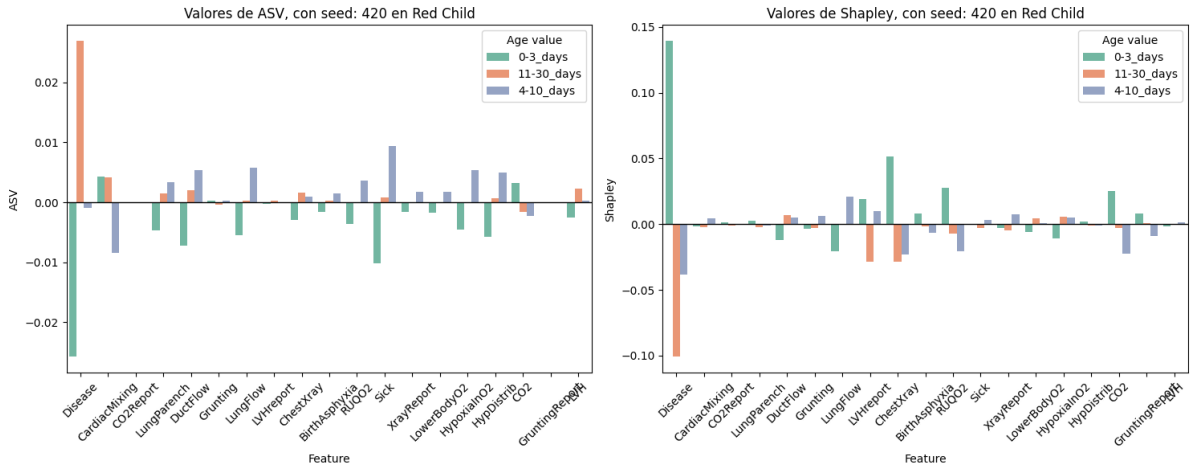


Figura 26: Resultados del ASV y Shapley para el modelo con mayor accuracy de los 5 seeds para la red *Child*, con un 68 % de accuracy.

8.3. ASV exacto sin EqClasses vs ASV aproximado

El objetivo de este experimento es comparar la performance de obtener los órdenes topológicos de un grafo que es un polytree, pero no es un *dtree*. Por lo tanto, solo los podemos obtener sampleándolos con nuestro Algoritmo 3 o generándolos con el algoritmo de Knuth, puesto que las clases de equivalencia solo las podemos obtener para los *dtrees*.

Cantidad de órdenes sampleados	Tiempo de sampleo (s)
100	0.9270
1000	1.1170
10000	4.7170
20000	7.7170
30000	10.8387

Tabla 2: Tiempos de ejecución para muestreo y generación de órdenes topológicos de la red *Child*

En la Tabla 2 podemos ver que el enfoque aproximado toma un tiempo tratable para samplear los órdenes. Aunque en nuestro análisis de la complejidad del mismo, habíamos llegado a una cota cuasi polinomial, el algoritmo se comporta de mejor manera en la práctica. También podemos observar que, como se mencionó al introducir el algoritmo de sampleo, su complejidad no es lineal en la cantidad de órdenes sampleados. Esto ocurre, pues vamos obteniendo múltiples candidatos en cada llamado recursivo de la función.

En base a estos resultados, uno podría creer que la mejor opción es generarlos con el algoritmo de Knuth, el cual tarda menos de 1 segundo en generar los 30000 órdenes. El problema de este algoritmo es que no respeta la distribución de los órdenes. Por ejemplo, si tuviéramos un grafo como el de la Figura 27, podría ocurrir que los primeros 1000 órdenes que nos devuelva el algoritmo tengan como primer nodo a n_1. Pero en realidad n_1 es el primer nodo en menos del 0.05 % de los casos, por ende no sería una muestra representativa. Para lograr esto deberíamos generar todos los órdenes, pero ya generar meramente un 1 % de los órdenes de la red *Child* tomaría más de 2 días.

Por último, realizamos un experimento para calcular el error al calcular *ASV* con el método aproximado, sampleando 1000 órdenes topológicos de la red *Child*. En la Figura 28 podemos ver los resultados de esta corrida. La mayoría de los valores aproximados obtenidos tiene un error del 8 % con respecto a su valor exacto. Las diferencias más altas se corresponden a valores de *ASV* muy pequeños, como 0.005, por lo que una pequeña diferencia en su valor calculado relativamente es más significativa. Esto es esperable, ya que el error mencionado en el Teorema 4 es un error absoluto, no relativo. Para los features con valores mayores a 0.01, su diferencia es menor al 5 %. Con estos resultados, podemos concluir que no es necesario obtener todos los ordenes y con una buena aproximación podemos obtener resultados medianamente precisos.

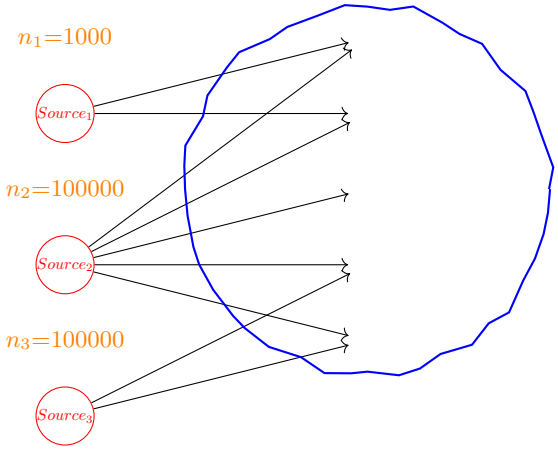


Figura 27: Posible comienzo del algoritmo exacto, con los candidatos a ser el primer nodo del orden en rojo y el resto del grafo en azul. Cada node fuente (source) tiene sus respectivas cantidades de órdenes topológicos en los que está primero.

8.4. Algoritmo de promedio para DT binarios

Para este experimento vamos a comparar la forma naive de obtener la predicción promedio para una permutación y el algoritmo 1, que utiliza la estructura del árbol para calcularlo con complejidad $O(i|V|+(varElim)l)$. La idea es comparar el tiempo que tardan ambos algoritmos y ver cómo se asemejan estos promedios a las probabilidades originales de la red. Dentro del cálculo de $Shap^{assym}(x, i)$, para la instancia x y el feature i, lo que queremos calcular es:

$$\nu(\pi) = \mathbb{E}_{N(x'|x)}[f_y(x_{\pi \leq i} \cup x'_{\pi > i})]$$

A través de una permutación π de los features de x definimos qué features quedan fijos y cuáles varían. La función de probabilidad que se utiliza es $N(x'|x)$, la cual utilizamos para calcular la probabilidad de los valores de x' dados los valores de x. Esta predicción promedio la vamos a calcular para cada uno de los posibles valores[25] y de p, el feature a predecir. $f_y(x)$ es un clasificador binario que devuelve 1 si nuestro árbol de decisión le asigna la clase y a la instancia x y 0 en el caso contrario. A continuación presentamos la implementación naive para calcular ν.

Para calcular $\mathbb{E}_{N(x'|x)}[f_y(x_{\pi \leq i} \cup x'_{\pi > i})]$ lo que vamos a hacer es generar todas las instancias $x_{prom} \in (x_{\pi \leq i} \cup x'_{\pi > i})$, en las cuales los valores de los features que aparece luego de x_i en π van a ser variables, y el resto van a ser fijos. Por fijos nos referimos a que van a tener los mismos valores que x, y sus otros features van a tomar todos los valores posibles. A partir de estas instancias vamos a calcular la función característica[26] cómo:

$$\nu(\pi) = \sum_{x_{prom} \in (x_{\pi \leq i} \cup x'_{\pi > i})} p_N(x_{prom}|x) f_y(x_{prom})$$

Por lo tanto, para esta cuenta vamos a necesitar generar todas las instancias $(x_{\pi \leq i} \cup x'_{\pi > i})$, y luego evaluar nuestro árbol de decisión DT y a la red bayesiana N para cada una de estas instancias. Evaluar él DT cuesta $O(d)$, siendo d la profundidad del DT. Si tomamos a c como la cardinalidad máxima de un feature y a $vars$ como el tamaño del conjunto de features variables, la complejidad temporal de la implementación naive es $O(vars^c(varElim + d)$, siendo $O(vars^c)$ la cantidad de instancias generadas.

Así las complejidades que nos quedan son $O(vars^c(varElim + d))$ y $O(i|V| + (varElim)l)$, para cada uno de nuestros algoritmos. Podemos ver que la solución naive depende de la cantidad de features variables del π, a diferencia del otro algoritmo, que corre el promedio directamente sobre la estructura del DT.

Para la red *Cancer* se generaron 600 instancias con un árbol de decisión de altura 3 y para la red *Child* se generaron 10000 instancias con un árbol de decisión de altura 9. Las probabilidades de la red bayesiana son los valores que devuelve la consulta $P(X = z)$ para la red N y los distintos valores z

[25] Recordemos que podemos tener variables no binarias, por lo que y puede tomar más valores que 0 o 1.

[26] La notación para la función característica es distinta a la utilizada al introducirla en la fórmula 2, pero su significado es el mismo. En este caso $(x_{\pi \leq i} \cup x'_{\pi > i})$ son nuestras instancias consistentes y f_y es nuestro clasificador binario M. Para cada valor de y, f_y devuelve 1 si la etiqueta clasificada es y y 0 en el caso contrario.

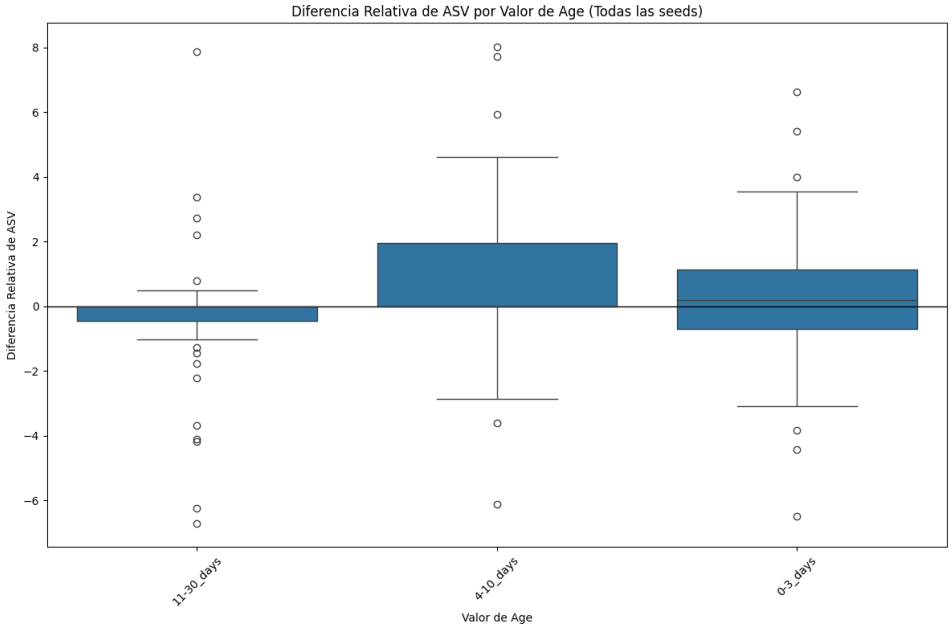


Figura 28: Diferencia relativa entre los valores del ASV aproximado y el exacto para la red *Child*, se utilizaron las diferencias de 30 seeds. Estos son los valores mayores a 0.01.

de cada feature X. El método *Algoritmo promedio (probabilidades)* consiste de la predicción promedio, si en el árbol en vez de devolver una predicción de 1 o 0 se devolviera una probabilidad en las hojas. Este método es distinto al algoritmo evaluado en esta tesis, pero nos pareció interesante agregarlo para analizar la diferencia entre los promedios al usar una predicción probabilística.

Método	**Predicción**	**Tiempo (segundos)**
Algoritmo promedio	[0.98255, 0.01745]	0.0165
Algoritmo promedio (probabilidades)	[0.7213, 0.2787]	0.0043
Implementación naive	[0.98255, 0.01745]	0.0043
Probabilidades red bayesiana	[0.7, 0.3]	-

Tabla 3: Valor promedio de la predicción del feature **Smoker** en la red bayesiana *Cancer*, dejando variables todos los features

Método	**Predicción**	**Tiempo (segundos)**
Algoritmo promedio	[0.9916, 0.0, 0.0084]	0.0258
Algoritmo promedio (probabilidades)	[0.7577, 0.0746, 0.1677]	0.0258
Implementación naive	[0.9916, 0.0, 0.0084]	19.6774
Probabilidades red bayesiana	[0.6490, 0.1715, 0.1795]	-

Tabla 4: Valor promedio de la predicción del feature **Age** en la red bayesiana *Child*, dejando variables 11 de los 20 features y utilizando a $x \in data$ t.q $f(x) = 0$.

Al analizar la Tabla 3 podemos ver que en ambos casos se tiende a sobrerrepresentar una clase. Esto ocurre ya que el modelo entrenado predice 0 para la mayoría de los inputs y la probabilidad de los inputs para los cuales predice 1 es más baja. En la Tabla 4 ocurre lo mismo respecto a la sobrerrepresentación. Por lo que, en realidad, la predicción promedio solo va a ser tan buena como el modelo que haya sido entrenado. Esto no depende del algoritmo, sino del entrenamiento del modelo. Además, podemos ver que la implementación naive es más lenta. Esto se debe a que la mediana de la cardinalidad de cada feature es 3. Por lo que cada feature agregado va a hacer que se tarde 3 veces más en promedio. Para órdenes topológicos que dejaran los 19 features variables la implementación naive tardaría **más de 1 día**. En cambio, la performance del algoritmo promedio no se ve tan afectada por la cantidad de

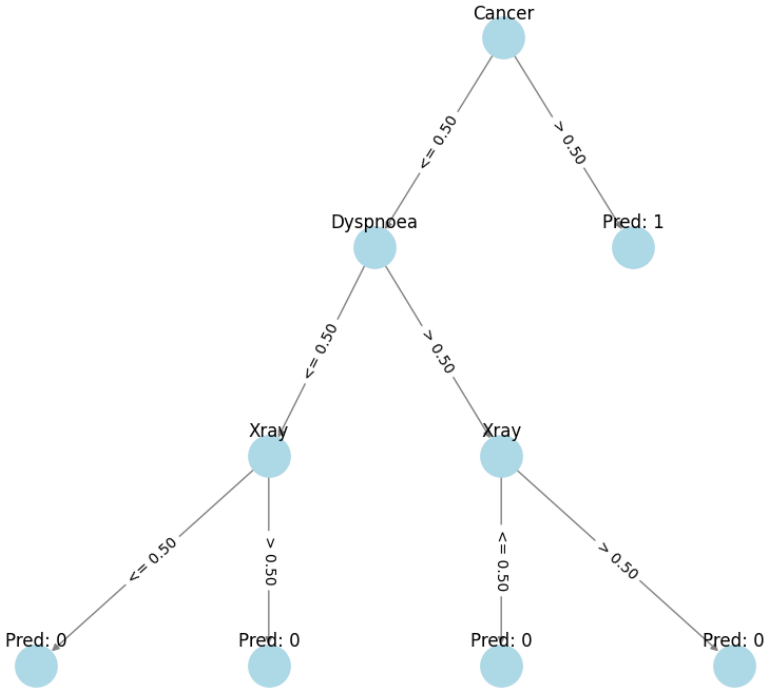


Figura 29: Árbol de decisión generado a partir de los datos de la red *Cancer*. Las hojas contienen el valor de la predicción que devuelve el modelo (0 o 1)

features variables, sino que depende del tamaño del árbol de decisión. Finalmente, se destaca que las predicciones más cercanas a las probabilidades generadas por la red bayesiana son aquellas que utilizan directamente las probabilidades como output, en lugar de predicciones binarias (0/1). Esto se debe a que estas predicciones son más granulares y, por ende, reflejan con mayor fidelidad las distribuciones probabilísticas subyacentes. Podemos ver en la Figura 29 que el patrón aprendido por el árbol es muy simple $f(x) =$ **If** $x_{cancer} > 0.5$, **then** 1, **else** 0. Luego como $P(Cancer = 1) = 0.01745$, el valor del promedio que vemos en la Tabla 3 va a representar esa predicción.

Se puede contemplar que los valores de la implementación naive y del algoritmo promedio son idénticos, puesto que ambos están calculando lo mismo. Solo que mientras nuestro algoritmo calcula la probabilidad de llegar a una hoja, la implementación naive genera todas las instancias que pueden llegar a la misma, para luego clasificarlas y calcular su probabilidad. Así que teniendo en cuenta que ambos calculan el mismo valor, podemos concluir que el algoritmo introducido ofrece una mejora significativa respecto a la implementación naive.

9. Conclusión

En el presente trabajo se optimizó el cálculo de las explicaciones brindadas a través de Assymetric Shapley Values (ASV), aplicadas a datos con distribuciones de redes bayesianas y modelos de árboles de decisión. Se demostró la tratabilidad del problema para el caso de la Naive Bayes y se buscó una solución polinomial para el caso más general.

Para el caso más general, se definió una noción de clase de equivalencia en base a los órdenes topológicos para disminuir la cantidad de evaluaciones de la predicción promedio. Para la predicción promedio en árboles de decisión, se implementó un algoritmo exacto más eficiente que la implementación naive. En el análisis teórico se identificó que el algoritmo diseñado para obtener las clases de equivalencia en árboles tiene una complejidad temporal polinomial respecto a la cantidad de clases de equivalencia.

Luego, se realizó un algoritmo aproximado para calcular ASV. Para el mismo, se llevó a cabo el cómputo del número de órdenes topológicos, el cual, en el caso general, se encuentra fuera de la clase de problemas polinomiales (#P-completo). No obstante, se mostró que para los polyforest con grado acotado (algo razonable en las redes bayesianas), es posible desarrollar algoritmos que logran una complejidad

tratable.

Desde el punto de vista práctico, se realizó la implementación de ASV utilizando las ideas desarrolladas en esta tesis. Se pudo observar que la cantidad de clases de equivalencia era menor que la cantidad de órdenes y que la heurística presentaba una mejora significativa. Aun así, dicha implementación resulta considerablemente más lenta en comparación con SHAP. Se debe optimizar la implementación de ASV, para lograr un desempeño que sea competitivo. La optimización podría orientarse hacia el uso de lenguajes más eficientes (como $C++$), técnicas de paralelización o encontrar nuevas heurísticas para calcular ASV. En lo que respecta al enfoque aproximado, para los grafos utilizados en la experimentación el problema era tratable y la aproximación tenía una precisión adecuada. Por último, al contrastarse los valores de SHAP y ASV, se encontró que ASV lograba identificar relaciones entre los datos que SHAP no contemplaba.

En síntesis, se obtuvo una implementación exacta de ASV para distintos modelos y distribuciones, con la posibilidad de expandir el framework a una mayor cantidad de familias. Además de un algoritmo de conteo y sampleo para órdenes topológicos en polytrees, el cual permite aproximar el valor de ASV. El aporte principal del trabajo es optimizar esta métrica a través de la heurística encontrada, la cual utiliza la noción de clases de equivalencia aplicada a los órdenes topológicos.

Trabajo futuro A continuación se detallan algunas líneas de trabajo futuro que surgieron a lo largo del desarrollo de esta tesis:

- **Generalizar el algoritmo de clases de equivalencia a polytrees:** adaptar el algoritmo actual, que solo admite árboles dirigidos, para que también funcione sobre estructuras más generales como *polytrees*, utilizando un enfoque similar al del conteo exacto de órdenes topológicos en estos grafos.
- **Optimizar el algoritmo de conteo:** investigar mejoras que permitan reducir la complejidad del algoritmo de conteo de órdenes topológicos, buscando una solución polinomial en el tamaño del grafo, independientemente del grado de sus vértices.
- **Extender el framework a modelos causales arbitrarios:** permitir el uso de modelos causales que no necesariamente sean redes bayesianas. Para ello, modificar el código para que pueda operar sobre modelos más generales, así como optimizar la ejecución mediante cacheo de resultados intermedios en las predicciones promedio.
- **Incorporar consultas de unión sobre evidencia parcial:** estudiar una implementación eficiente de la operación de unión de consultas en redes bayesianas mediante *Variable Elimination*, aprovechando la posibilidad de reutilizar eliminaciones intermedias. Esta operación resulta útil al introducir evidencia parcial sobre variables y la realizamos múltiples veces en el algoritmo de predicción promedio. En bnlearn se implementa esta consulta, pero está implementada como una suma de probabilidades.
- **Estudiar propiedades de complejidad de órdenes topológicos:** formalizar y demostrar si existe una equivalencia entre la posibilidad de contar órdenes topológicos en tiempo polinomial y muestrearlos eficientemente en dicho tiempo.
- **Explorar algoritmos alternativos para enumerar órdenes topológicos:** implementar y adaptar ideas provenientes de trabajos previos, como [**?**], con el objetivo de acelerar el muestreo de órdenes topológicos en nuestro enfoque.
- **Implementar nuevas estrategias de sampleo:** durante la investigación de trabajos similares realizados encontramos un algoritmo de sampleo [**?**] y un algoritmo de conteo [**?**] para órdenes topológicos. Al implementar estos algoritmos podríamos utilizarlos para mejorar la complejidad de nuestra solución.

Referencias

10. Apéndice

10.1. Fórmulas

10.1.1. Funciones auxiliares del cálculo de los unrelated trees

Está es la definición de la función *union* utilizada en la Ecuación 6:

$$union(((repEC_1, lTopo_1, rTopo_1), ..., (repEC_{|n|}, lTopo_{|n|}, rTopo_{|n|})), n_t) =$$
$$\left(\bigcup_{j=1}^{|n|} repEC_j \cup \{n_t\}, \binom{\sum_{i=1}^{|n|} |L(repEC_i)|}{|L(repEC_1)|, \ldots, |L(repEC_{|n|})|} \prod_{i=1}^{|n|} lTopo_i, \binom{\sum_{i=1}^{|n|} |R(repEC_i)|}{|R(repEC_1)|, \ldots, |R(repEC_{|n|})|} \prod_{i=1}^{|n|} rTopo_i \right) \tag{10}$$

La función *union* está definida en la Ecuación 10, y es la encargada de unir las distintas clases de equivalencia de los hijos de un nodo n. Más formalmente, $union(n, ((repEC_1, lTopo_1, rTopo_1), \ldots)$ es la clase de equivalencia que se representa con $repEC$ en la cual el nodo n está a la izquierda de x_i y es compatible con las clases $repEC_i$ (en el sentido de que si un nodo aparece a la izquierda en $repEC_i$ entonces este también aparece a la izquierda en $repEC$, y de la misma forma con los que aparecen a la derecha). Para calcular la cantidad de órdenes topológicos a la izquierda y a la derecha de la clase utilizamos una fórmula muy similar a 4.3. Se puede aplicar la misma lógica porque no hay dependencias entre los nodos de los árboles no relacionados y sus subárboles, por lo que podemos combinar los órdenes topológicos sin restricciones.

Complejidad temporal de *union* *union* realiza 4 operaciones de costo $O(n)$ (pues $d_{out}(node) < n$), $2 \prod$ y $\sum$. El coeficiente multinomial[27] tiene costo $O(n)$, por lo que *union* tendrá una complejidad temporal de $O(n)$.

10.1.2. Función completa de #LO

A continuación vamos a ver la implementación del algoritmo descrito en la sección 2. Los parámetros de la función serán:

- p: posición en la que se colocó el último ancestro.
- i: índice del ancestro que vamos a colocar.
- $nodesPerAncestor$: una lista que contiene en la i-ésima posición el número de nodos a la izquierda del unrelated tree de a_i.
  - Debido a la ligera mejora mencionada en la sección 5.2.1, solo habrá un árbol debajo de cada a_i, ya que si hay más de uno, estos árboles serán fusionados en uno solo.

Estas son las funciones auxiliares que vamos a utilizar:

- possibleCombinations$(l, i, nodesPerAncestor)$ o pb, que devuelve todas las formas posibles de sumar l, utilizando los nodos de los primeros i elementos de $nodesPerAncestor$.
  - Por ejemplo, possibleCombinations$(5, 3, [4, 2, 1, 7, \ldots]) = [[4, 0, 1], [4, 1], [3, 1, 1], [3, 2], [2, 2, 1]]$.
- hasPlaced$(placedNodes, nodesPerAncestor)$ o hp, que devuelve una lista l que tiene en la posición i-ésima el elemento $l[i] = nodesPerAncestor[i] - placedNodes[i]$. Lo que hace es restar de $nodesPerAncestor$ los elementos que ya colocamos en $placedNodes$.
- canPlace$(i, nodesPerAncestor)$ o cp, que devuelve la suma de los primeros i elementos de $nodesPerAncestor$.

Teniendo en cuenta estas funciones, la fórmula[28] que vamos a utilizar para calcular los posibles ordenamientos de los nodos de las distintas clases de equivalencia de cada subárbol es:

$$\#LO(p, i, npa) = \begin{cases} \binom{cp(i, npa)}{npa[1], \ldots, npa[i]} & \text{if } |A| = i \\ \sum_{toFill=p}^{p+cp(i,npa)} \sum_{comb \in pb(toFill-p-1, i, npa)} \left(\binom{sum(comb)}{comb_1, \ldots, comb_i} \right. & \\ \left. \cdot \#LO(toFill, i+1, hp(comb, npa)) \right) & \text{otherwise} \end{cases}$$

[27] Utilizamos el modelo de memoria RAM teniendo en cuenta que los factoriales pueden estar precalculados y que multiplicar números es $O(1)$ más allá de su tamaño. Igualmente, si el multinomial tomara $O(n^2)$, la complejidad total seguiría siendo la misma.

[28] Pueden encontrar este algoritmo en `\pasantia-BICC\asvFormula\classesSizes\recursiveFormula.py`

Nuestro caso base es cuando ya hemos colocado todos los ancestros, por lo tanto solo queda combinar los nodos que nos quedan por colocar. Luego, en el caso recursivo, él $toFill - p - 1$ es el número de posiciones que necesitamos llenar entre el a_i que estamos colocando y el a_{i-1} que se colocó antes. En cada paso realizamos la sumatoria de cada posible combinación *comb* de los elementos de *npa*, teniendo en cuenta todas las posibilidades de la cantidad de posiciones a llenar $toFill$ entre a_i y a_{i-1}. Las posiciones van desde p hasta p más la máxima cantidad de nodos que podemos colocar ($cp(i, npa)$). Ahora, con todas estas funciones definidas, la llamada que resolverá $leftOrders(A, left)$ será $\#LO(0, 0, left)$.

Para entender un poco cómo funciona este algoritmo veamos cuál podría ser un paso del mismo. En la Figura 30 podemos ver cómo sería el estado actual de los órdenes topológicos que estamos contando, teniendo en cuenta las decisiones previas que tomamos. Los nodos pintados en naranja que vemos en la Figura 31 son los nodos que tenemos disponibles para rellenar ese espacio en rojo. En cambio, los nodos de u_i no los podemos utilizar todavía, ya que todavía no colocamos el nodo a_i. Una vez que lo coloquemos podremos utilizarlos.

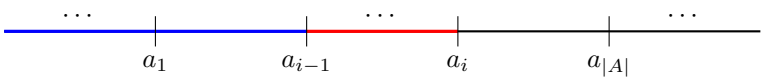


Figura 30: i-ésimo paso de #LO. La línea azul son los nodos ya colocados y la línea roja tiene longitud $toFill-p-1$, definiendo el espacio que hay para colocar los nodos disponibles.

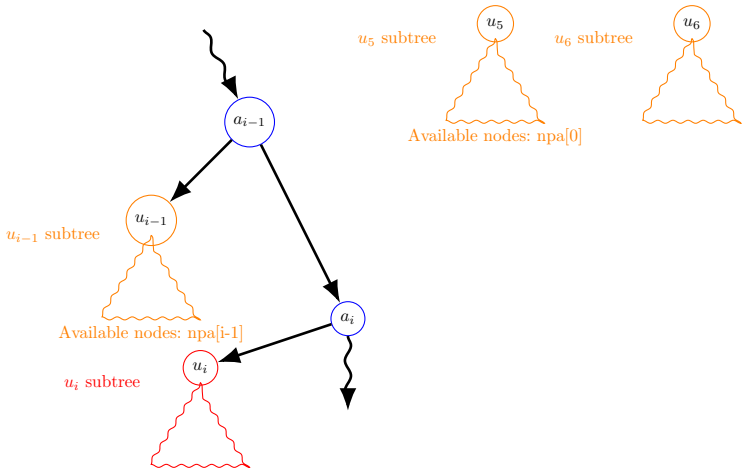


Figura 31: Los nodos pintados en naranja son los nodos disponibles para ser colocados en el paso i. Para cada conjunto de subárboles *npa* tiene la cantidad de nodos disponibles.

Complejidad temporal de $\#LO$ Analicemos la complejidad temporal de $\#LO$, para obtenerla necesitamos saber cuánto cuesta calcular cada nodo y cuántos estados posibles hay.

Comencemos con los estados posibles. Tenemos los parámetros i, p y $nodesPerAncestor$. El índice i tiene $|A|$ valores posibles, con $|A| \in O(n)$. Luego p puede tomar $O(n)$ valores posibles, debido a que un orden topológico puede tener como máximo n nodos. Ahora tenemos que calcular el número de estados posibles para $nodesPerAncestor$.

Para cada posición $npa[i]$ vamos a tener el número de nodos a la izquierda de los unrelated trees bajo a_i. Definimos $size(ut_i)$ el tamaño del unrelated tree ut_i bajo el nodo a_i (si no hay ninguno, es 0). Para la posición i-ésima, tenemos $size(ut_i)$ valores posibles. Por la Fórmula 1, sabemos que $\#EC(ut_i) > size(ut_i)$, por lo que los valores posibles de $nodesPerAncestor$ están acotados por $\prod_{i=0}^{|A|} size(ut_i) < \prod_{i=0}^{|A|} \#EC(ut_i)$ (esta es la combinatoria de cada valor posible en cada posición). Utilizando la Fórmula 1, sabemos que $\prod_{i=0}^{|A|} \#EC(ut_i) < |equivalenceClasses|$, ya que la productoria de las clases de cada subárbol no puede ser mayor a la cantidad total de clases. Con eso, podemos concluir que los valores posibles de *npa están acotados por el número de clases de equivalencia.*

Entonces, concluimos que el número de estados de #LO es $values(i) \cdot values(p) \cdot values(npa) \in O(n) \cdot O(n) \cdot |equivalenceClasses| = O(n^2 \cdot |equivalenceClasses|)$.

Queda ver cuánto cuesta computar cada nodo. El coeficiente multinomial toma $O(n^2)$, así que el número de sumas determinará nuestra complejidad. $toFill$ puede tomar como máximo $\sum_{j=0}^{i} nodesPerAncestor[j] \in$

$O(n)$ valores en cada iteración.

Resta determinar cuántos valores puede tomar possibleCombinations$(l, i, nodesPerAncestor)$. Sabemos que para cada resultado res tal que $res \in pb(l, i, npa)$ se cumple que $\forall 0 < j < |res|, 0 < res[j] < npa[j]$. Usando un argumento similar al de $nodesPerAncestor$, podemos concluir que $size(pb(l, i, npa)) < |equivalenceClasses|$. Así concluimos que el costo de calcular cada estado es $O(n \cdot |equivalenceClasses| \cdot n^2) = O(n^3|equivalenceClasses|)$, que es la cantidad de iteraciones de la primera y segunda sumatoria, multiplicado por el costo de evaluar el multinomial. Por lo que finalmente la complejidad de calcular $leftSize$ es $O(|estados|) \cdot O(computarNodo) = O(n^5 \cdot |equivalenceClasses|^2)$.

10.1.3. Funciones auxiliares del cálculo de clases de equivalencias

$$eqClass(A, D, ((eqCl_1, \_, \_), ..., (eqCl_{|UR|}, \_, \_))) = \{a_l \mid a \in A\} \cup \{d_r \mid d \in D\} \cup (\bigcup_{j=1}^{|UR|} eqCl_j)$$

$$eqClassTopos(A, D, mix) = leftSize(A, mix) \cdot rightSize(D, mix)$$

$$leftSize(A, ((,l_1, \_), ..., (,l_{|UR|}, \_))) = leftOrders(A, [l_1, \ldots, l_{|UR|}]) \cdot \prod_{i=1}^{|UR|} l_i$$

$$rightSize(D, ((eqCl_1, \_, r_1), ..., (eqCl_{|UR|}, \_, r_{|UR|}))) = \binom{|D| + \sum_{i=1}^{|n|} |R(eqCl_i)|}{|R(eqCl_1)|, \ldots, |R(eqCl_{|n|})|, |D|} \cdot \prod_{i=1}^{|UR|} r_i * \#topos(x_i)$$

En $eqClass$ obtenemos representación de la clase de equivalencia $\mathsf{Rep}([\pi]_R)$, colocando los nodos en A antes de x_im los nodos en D después y dejando con el mismo tag los nodos de mix. En $\#eqClass$ obtenemos el tamaño de la clase de equivalencia, multiplicando todos los órdenes topológicos posibles de la izquierda por los de la derecha, ya que una combinación de ambos respetará la clase de equivalencia. Para $rightSize$ usamos la misma fórmula que antes y simplemente añadimos los órdenes topológicos de los descendientes utilizando la fórmula de 4.3. Y para $leftSize$ calculamos las posibles combinaciones usando la función que aparece a continuación.

Complejidad de eqClassTopos Primero, necesitamos calcular la complejidad temporal de $rightSize$, que es $O(n^2)$. Tenemos el coeficiente multinomial, que toma $O(n)$ tiempo , el $\prod$ que es $O(n)$, ya que $|UR| < n$ y $\#topos(x_i)$ que puede implementarse en $O(n^2)$. Entonces, la suma de estas operaciones tiene complejidad $O(n^2)$.

Luego para $leftSize$ sabemos que la complejidad temporal del Algoritmo 2 es de $O(n^5 \cdot |equivalenceClasses|^2)$. En la sección 10.1.2 del apéndice se encuentra la justificación de esta complejidad.

Eso nos deja con una complejidad temporal de $O(n^5 \cdot |equivalenceClasses|^2)$ para $eqClassSizes$, puesto que la complejidad de $rigthSize$ está acotada por la de $leftSize$.

10.1.4. Complejidad de `allPossibleOrders`

El cálculo para obtener la complejidad de $allPosibleOrders$ es muy similar al hecho en 10.1.3 para #LO, solo que ahora no tenemos las clases de equivalencia para acotar los valores. Así que tenemos que hacer las cuentas combinatorias para tener una complejidad en función del digrafo D. Al también usar dinámica vamos a tener que calcular la cantidad de estados y cuánto cuesta computar cada uno.

Los valores que tenemos que tener en cuenta para nuestros estados son $nodeIndex, nodesBefore, nodesAfter$.

El algoritmo `allPossibleOrders` utiliza programación dinámica con memoización. Su complejidad total se obtiene al multiplicar la cantidad de *estados* posibles por el *costo computacional por estado.*

Sean:

- $k = d_{\text{in}}(v) + d_{\text{out}}(v) + 1$ el número total de vecinos (padres e hijos) del nodo actual más el propio nodo.
- S la cantidad total de nodos a colocar, es decir, $S =$ `nodosRestantes` $- k$.

Número de estados Cada estado está definido por un índice $i \in [0, k]$ y dos vectores de tamaño k: `nodesBefore`, `nodesAfter`, con entradas en $\mathbb{N}$ cuya suma total es S. La cantidad de formas de repartir S unidades entre $2k$ casillas (coordenadas de los vectores `nodesBefore` y `nodesAfter`) es:

$$\binom{S+2k-1}{S}$$

Entonces, el número total de estados es $O\left(k \cdot \binom{S+2k-1}{S}\right)$.

Costo por estado Dentro de cada estado se generan todas las combinaciones posibles de distribuir $t \leq S$ nodos[29] entre hasta k posiciones. Para cada t hay $\binom{t+k-1}{k-1}$ combinaciones posibles. Luego por la identidad de la escalera se cumple que:

$$\sum_{t=0}^{S} \binom{t+k-1}{k-1} = \binom{S+k}{k},$$

Cada una de estas combinaciones se procesa en $O(k)$. Por lo tanto, el costo por estado es: $O\left(\binom{S+k}{S} \cdot k\right)$.

Complejidad total Multiplicando la cantidad de estados por el costo por estado:

$$T(S,k) = O\left(k^2 \cdot \binom{S+2k-1}{S} \cdot \binom{S+k}{S}\right).$$

Casos particulares A continuación analizamos cómo se comporta la complejidad en función de los parámetros S y k:

- **Caso 1: k constante (por ejemplo, $k = 3$).** Si el número de vecinos involucrados en la combinación es una constante fija, entonces los binomios

  $$\binom{S+2k-1}{S} \quad \text{y} \quad \binom{S+k}{S}$$

  se comportan como polinomios en S de grado $2k-1$ y k respectivamente. Por lo tanto, la complejidad total se acota por un polinomio:

  $$T(S) = O\left(S^{3k-1}\right),$$

  lo que resulta eficiente para instancias en las que el número de vecinos a combinar está acotado.

- **Caso 2: $k = O(n)$ y $S = O(n)$.** En el peor caso, donde el nodo actual tiene un número lineal de vecinos (por ejemplo, si el grafo es muy denso o el nodo es central en la topología), tanto k como S pueden crecer linealmente con n. En este escenario, los coeficientes binomiales se comportan asintóticamente como:

  $$\binom{S+2k-1}{S} = \Theta\left(\frac{(3n)!}{n! \cdot (2n)!}\right), \qquad \binom{S+k}{S} = \Theta\left(\frac{(2n)!}{n! \cdot n!}\right),$$

  lo cual implica una complejidad exponencial. En particular:

  $$T(n) = O\left(n^2 \cdot \binom{3n}{n} \cdot \binom{2n}{n}\right),$$

  que es significativamente mayor que $n!$ y confirma que el algoritmo no es eficiente en este caso.

[29] En realidad no se tiene en cuenta hasta S, sino meramente los nodos que se pueden colocar en ese momento, pero utilizamos S como cota.

10.2. Demostraciones

10.2.1. ASV puede calcularse en tiempo polinomial para una distribución Naive Bayes

Teorema 2. *Los Asymmetric Shapley Values pueden calcularse en tiempo polinomial para distribuciones dadas como una Red Bayesiana Naive y para una familia de modelos $\mathcal{F}$ si y solo si los Shapley values pueden calcularse para la familia $\mathcal{F}$ bajo una distribución producto arbitraria en tiempo polinomial.*

Demostración. Primero, probamos la implicación de derecha a izquierda. Sea x_1 el padre de todos los demás nodos en el DAG. Vamos a mostrar cómo calcular $Shap^{assym}_{M,e,\Pr}(x_j)$ para cualquier $2 \leq j \leq n$ y $Shap^{assym}_{M,e,\Pr}(x_1)$ de forma independiente.

Observemos que el DAG tiene $(n-1)!$ órdenes topológicos, uno para cada permutación de los features $\{x_2, \ldots, x_n\}$, y $\pi(x_1) = 1$ para todas ellas. Entonces,

$$Shap^{assym}_{M,e,\Pr}(X_j) = \sum_{\pi \in topos(G)} w(\pi) \left[\nu_{M,e,\Pr}(\pi_{<j} \cup \{x_j\}) - \nu_{M,e,\Pr}(\pi_{<j})\right]$$

es equivalente a:

$$Shap^{assym}_{M,e,\Pr}(X_j) = \frac{1}{(n-1)!} \sum_{\pi \in perm(\{x_2,\ldots,x_n\})} \left[\nu_{M,e,\Pr}(\{x_1\} \cup \pi_{<j} \cup \{x_j\}) - \nu_{M,e,\Pr}(\{x_1\} \cup \pi_{<j})\right]$$

Así, una vez que x_1 está fijado, la distribución para las variables $x_2, \ldots, x_n$ es una distribución producto con $p_{x_j} = P(X_j = 1 | X_1 = e(x_1))$. Para simplificar, asumamos que $e(x_1) = 1$, y consideremos la distribución de producto $\Pr'$ definida como:

$$\Pr{}'[X_i = 1] = p_i = \begin{cases} 1 & i = 1 \\ \Pr[X_i = 1 | X_1 = e(x_1)] & \text{en otro caso} \end{cases}$$

que intuitivamente se obtiene de Pr fijando $X_1 = 1$. Siempre que x_1 esté fijo, ambas distribuciones Pr y $\Pr'$ se comportan de la misma forma:

Lema 2. *Para cualquier $S \subseteq X \setminus \{x_1\}$, se cumple que:*

$$\nu_{M,e,\Pr}(\{x_1\} \cup S) = \nu_{M,e,\Pr'}(\{x_1\} \cup S) = \nu_{M,e,\Pr'}(S)$$

Demostración. Esto se sigue de directamente manipulando la expresión:

$$\begin{aligned}
\nu_{M,e,\Pr}(\{x_1\} \cup S) &= \sum_{e' \in \mathtt{cw}(e, \{x_1\} \cup S)} \Pr[e'|S] M(e') \\
&= \sum_{e' \in \mathtt{cw}(e, \{x_1\} \cup S)} \left(\prod_{\substack{e'(y)=1 \\ y \notin \{x_1\} \cup S}} p_y \prod_{\substack{e'(y)=0 \\ y \notin \{x_1\} \cup S}} (1 - p_y) \right) M(e') \qquad (11) \\
&= \sum_{e' \in \mathtt{cw}(e, S)} \left(\prod_{\substack{e'(y)=1 \\ y \notin S}} p_y \prod_{\substack{e'(y)=0 \\ y \notin S}} (1 - p_y) \right) M(e') \\
&= \sum_{e' \in \mathtt{cw}(e, S)} \Pr{}'[e'|S] M(e') \\
&= \nu_{M,e,\Pr'}(S)
\end{aligned}$$

donde la segunda igualdad viene de reemplazar Pr por la distribución producto y la tercera igualdad se deduce al observar que para todas las entidades $e' \in \mathtt{cw}(e, S)$ tales que $e'(x_1) = 0$ se cumple que

$$\prod_{\substack{e'(y)=1 \\ y \notin S}} p_y \prod_{\substack{e'(y)=0 \\ y \notin S}} (1 - p_y) = 0$$

Además, la segunda ecuación es igual a $\nu_{M,e,\Pr'}(\{x_1\} \cup S)$. □

Usando este lema, ahora demostramos

$$Shap^{assym}_{M,e,\Pr}(x_j) = Shap_{M,e,\Pr'}(x_j) \tag{12}$$

Se cumple que

$$\begin{aligned}
Shap_{M,e,\Pr'}(x_j) &= \frac{1}{n!} \sum_{\pi \in perm(X)} [\nu_{M,e,\Pr'}(\pi_{<j} \cup \{x_j\}) - \nu_{M,e,\Pr'}(\pi_{<j})] \\
&= \frac{1}{n!} \sum_{\pi \in perm(X)} [\nu_{M,e,\Pr'}(\{x_1\} \cup \pi_{<j} \cup \{x_j\}) - \nu_{M,e,\Pr'}(\{x_1\} \cup \pi_{<j})] \\
&= \frac{n}{n!} \sum_{\pi \in perm(\{x_2,\ldots,x_n\})} [\nu_{M,e,\Pr'}(\{x_1\} \cup \pi_{<j} \cup \{x_j\}) - \nu_{M,e,\Pr'}(\{x_1\} \cup \pi_{<j})] \\
&= \frac{1}{n-1!} \sum_{\pi \in perm(\{x_2,\ldots,x_n\})} [\nu_{M,e,\Pr}(\{x_1\} \cup \pi_{<j} \cup \{x_j\}) - \nu_{M,e,\Pr}(\{x_1\} \cup \pi_{<j})] \\
&= Shap^{assym}_{M,e,\Pr}(x_j)
\end{aligned}$$

donde la segunda y cuarta igualdad se siguen del Lema 2, la última igualdad de la Ecuación 10.2.1 y la tercera al observar que para cada permutación de $perm(\{x_2, \ldots, x_n\})$ podemos construir n permutaciones de $perm(X)$ insertando x_1 en todos los lugares posibles, y que para cada una de estas permutaciones la expresión dentro de la suma es la misma. Así, la Ecuación 12 muestra que calcular $Shap^{assym}_{M,e,\Pr}(x_j)$ se reduce a calcular los valores Shapley habituales para una distribución independiente particular.

Ahora consideramos $Shap^{assym}_{M,e,\Pr}(x_1)$. Observemos que

$$\begin{aligned}
Shap^{assym}_{M,e,\Pr}(x_1) &= \frac{1}{|topos(G)|} \sum_{\pi \in topos(G)} [\nu_{M,e,\Pr}(\pi_{<1} \cup \{x_1\}) - \nu_{M,e,\Pr}(\pi_{<1})] \\
&= \frac{1}{|topos(G)|} (\nu_{M,e,\Pr}(\{x_1\}) - \nu_{M,e,\Pr}(\emptyset))
\end{aligned}$$

ya que para todas las permutaciones el conjunto $\pi_{<1} = \emptyset$. Además, sabemos que $\nu_{M,e,\Pr}(\{x_1\}) = \nu_{M,e,\Pr'}(\emptyset)$ por el Lema 2, y que

$$\begin{aligned}
\nu_{M,e,\Pr}(\emptyset) &= \sum_{e' \in \mathtt{ent}(X)} \Pr[e] M(e) \\
&= \sum_{e' \in \mathtt{ent}(X)} \sum_{e'(x_1)=1} \Pr[e'|e'(x_1) = 1] \Pr[X_1 = 1] M(e') + \sum_{\substack{e' \in \mathtt{ent}(X) \\ e'(x_1)=0}} \Pr[e'|e'(x_1) = 1] \Pr[X_1 = 0] M(e') \\
&= \Pr[X_1 = 1] \nu_{M,e,\Pr'}(\emptyset) + \Pr[X_1 = 0] \nu_{M,e,\Pr''}(\emptyset)
\end{aligned}$$

donde la distribución $\Pr''$ es la distribución producto obtenida al fijar $X_1 = 0$ como

$$\Pr{}''[X_i = 1] = \begin{cases} 0 & i = 0 \\ Pr[X_i = 1 | X_1 = 0] & \text{de otro modo} \end{cases}$$

Finalmente,

$$Shap^{assym}_{M,e,\Pr}(x_1) = \frac{1}{|topos(G)|} ((1 - \Pr[X_1 = 1]) \nu_{M,e,\Pr'}(\emptyset) - \Pr[X_1 = 0] \nu_{M,e,\Pr''}(\emptyset))$$

y reducimos el problema de calcular $Shap^{assym}_{M,e,\Pr}(x_1)$ al de calcular la predicción promedio del modelo M para dos distribuciones independientes diferentes $\Pr'$ y $\Pr''$. Según [**?**][Teorema 1], se cumple que estos promedios pueden ser calculados en tiempo polinomial si y sólo si los valores de Shapley pueden ser calculados en tiempo polinomial para una distribución de producto arbitraria.

La demostración de izquierda a derecha sigue al observar que una distribución producto es un caso particular de una Distribución Naive Bayes en la cual, para cualquier valor del padre X_1, las distribuciones condicionales son las mismas.

□

10.2.2. Cota superior para la cantidad de clases de equivalencia

Lema 3. *Para cualquier árbol T, donde l es el número de hojas y h es la altura para la raíz n de T. Entonces, para la fórmula:*

$$\#EC(n) = \begin{cases} 2 & \text{if } n \text{ is a leaf} \\ \prod_{c \in children(n)} \#EC(c) + 1 & \text{oc.} \end{cases}$$

Tenemos la cota $\#EC(n) \leq h^l + 1$

Demostración. Queremos demostrar que $\#EC(n) \leq h_n^{l_n} + 1$. Esto aplica a cualquier árbol T y su raíz n, donde l_n es el número de hojas del árbol y h_n es su altura.

Debemos demostrar esto para los dos casos que presenta la fórmula.

Caso 1: n es una hoja Si n es una hoja, entonces tiene altura 1, y su número de hojas es 1. Esto nos deja con $\#EC(n) = 2 \leq 2 = (1)^1 + 1$.

Caso 2: n tiene hijos

Nuestra *hipótesis inductiva* es que para cada subárbol T_h que tiene un hijo c de n como raíz, nuestra fórmula se cumple. Esto significa que $\forall c \in children(n), \#EC(c) \leq h_c^{l_c} + 1$. Con esto en mente, veamos que

$$\begin{aligned} \#EC(n) &= (\prod_{c \in children(n)} \#EC(c)) + 1 \\ &\leq (\prod_{c \in children(n)} h_c^{l_c} + 1) + 1 \\ &\leq \prod_{c \in children(n)} h_n^{l_c} + 1 \\ &= h_n^{\sum_{c \in children(n)} l_c} + 1 \\ &= h_n^{l_n} + 1 \end{aligned}$$

En la primera desigualdad aplicamos la hipótesis inductiva. Después, en la segunda desigualdad sabiendo que por la definición de altura se cumple $(\forall c \in children(n))\ h_n = h_c + 1$, entonces $\forall k \in \mathbb{N}$ se cumple $h_n^k = (h_c + 1)^k \geq h_c^k + 1$. En la última igualdad utilizamos que $\sum_{c \in children(n)} l_c = l_n$).

Así concluimos que nuestra fórmula tiene una cota superior de $h^l + 1$. □

10.2.3. Error para la estimación de ASV a través de sampleos

El siguiente lema nos permite ver que dado un mecanismo de sampleo de este tipo sería posible aproximar el ASV de forma eficiente.

Lema 4 (Estimación de ASV a través de sampleos)**.** *Sea M un modelo, e una entidad, G un grafo causal para el conjunto de features X de M y* Pr *una distribución sobre el conjunto de entidades. Supongamos que existe un algoritmo $\mathcal{A}$ que dado un subconjunto $S \subseteq X$ de features, calcula el valor $\nu_{M,e,\Pr}(S) = \mathbb{E}[M(e')|\mathsf{cw}(e, S)]$. Supongamos también que existe un mecanismo $\mathcal{D}$ para samplear órdenes topológicos de G de forma uniforme.*

Luego, dada una precisión $\varepsilon > 0$ y una probabilidad $\delta \in (0, 1)$, es posible estimar el ASV para un modelo M, una entidad e y una feature x con precisión ε y probabilidad de error $1 - \delta$ usando $O(\log(1-\delta)/\varepsilon^2)$ sampleos de $\mathcal{D}$ y $O(\log(1-\delta)/\varepsilon^2)$ llamados a $\mathcal{A}$.

Demostración. Veamos a $\mathcal{D}$ como una variable aleatoria distribuida uniformemente sobre

$$T = \{\overline{x} \in \pi(X) : \overline{x} \text{ es un orden topológico de } G\}$$

(es decir, sobre los órdenes topológicos de G). Luego, si definimos $g : T \to \mathbb{R}$ como

$$g(\overline{x}) = \nu_{M,e,\Pr}(\overline{x}_{<x} \cup \{x\}) - \nu_{M,e,\Pr}(\overline{x})$$

tenemos que

$$ASV(M, e, x) = \frac{1}{|T|} \sum_{\overline{x} \in T} g(\overline{x}) = \mathbb{E}[g(\mathcal{D})] \tag{13}$$

Notemos que podemos samplear la variable $g(\mathcal{D})$ sampleando primero $\mathcal{D}$ y luego usando el algoritmo $\mathcal{A}$.

La ecuación (13) nos dice que podemos estimar el ASV estimando el valor $\mathbb{E}[g(\mathcal{D})]$, el cual a su vez podemos estimar sampleando la variable $g(\mathcal{D})$. Para obtener una cota a la cantidad de sampleos necesarios para obtener precisión ε usaremos desigualdades tradicionales, como la de Hoeffding. La misma dice que dadas variables aleatorias $\{X_i\}_{1 \leq i \leq k}$ independientes e idénticamente distribuidas con valores en el rango (a, b) se tiene que

$$P(|\sum_{i=1}^{k} X_i - \mathbb{E}[X_i]| \geq \varepsilon) \leq 2e^{-2\varepsilon^2 k/(b-a)}$$

En nuestro caso, la variable $g(\mathcal{D})$ tiene valor en el rango $[-1, 1]$, por lo que podemos pedir que la probabilidad de error sea a lo sumo $1 - \delta$ despejando la ecuación

$$2e^{-\varepsilon^2 k} \leq 1 - \delta$$

, de lo cual se obtiene que

$$\frac{\log((1-\delta)/2)}{e^2} \leq k$$

Por lo tanto, haciendo $k = O(\log(1-\delta)/\varepsilon^2)$ sampleos se obtiene un estimador con precisión ε y probabilidad δ.

□

10.3. Figuras

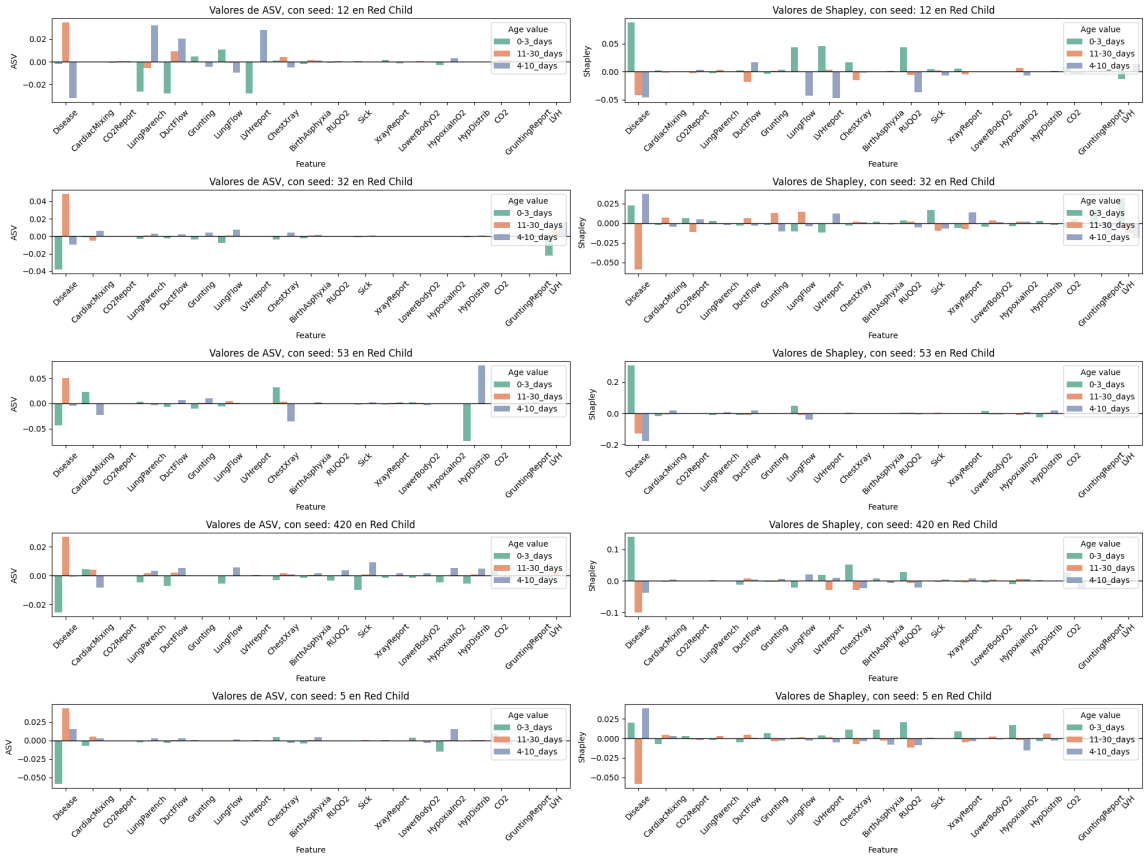


Figura 32: Comparación de los valores de ASV y Shapley para distintas seeds para la red Child

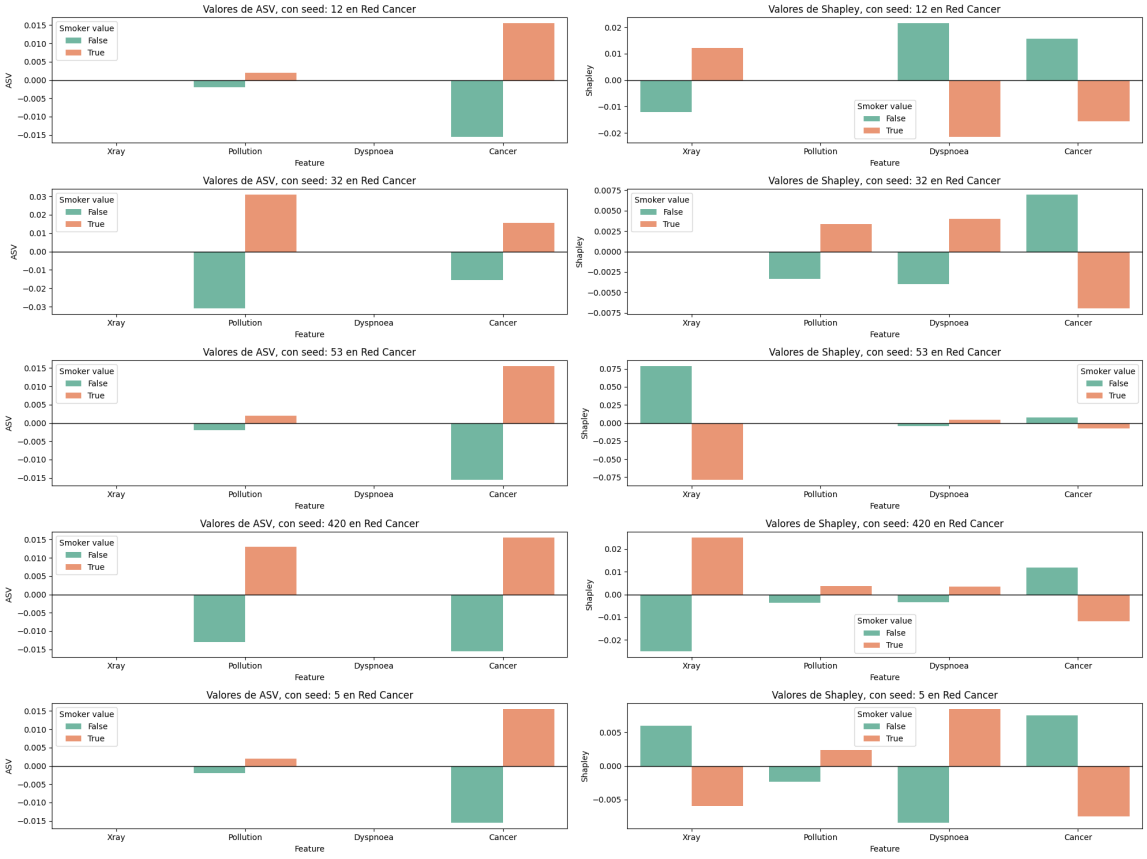


Figura 33: Comparación de los valores de ASV y Shapley para distintas seeds para la red Cancer