Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Máster Universitario en Ingeniería Informática

Trabajo Fin de Máster

Síntesis de explicaciones en tablas de decisiones óptimas

Autor: Enrique Jiménez Fernández
Tutor: Juan Antonio Fernández del Pozo de Salamanca

Madrid, junio de 2023

Este Trabajo Fin de Máster se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Máster
Máster Universitario en Ingeniería Informática

Título: Síntesis de explicaciones en tablas de decisiones óptimas

junio de 2023

Autor: Enrique Jiménez Fernández
Tutor: Juan Antonio Fernández del Pozo de Salamanca
Departamento de Inteligencia Artificial
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

La toma de decisiones es una asignatura que se ha ido tornando cada vez más difícil con los años y no es más que un reflejo de la complejidad del mundo en el que vivimos y la velocidad a la que cambia.

Es por ello que hace unas décadas se crearon los sistema de ayuda de toma de decisión (DSS). Estos permiten un marco de análisis y evaluación de las políticas de decisión metodológico y eficiente, ahorrando así esfuerzo a las personas a la hora de tomar decisiones.

Pero, aun así y a causa de la propia complejidad de las bases de conocimiento que modelizan los problemas de decisión, es necesario utilizar estructuras que sinteticen el conocimiento sin alterar el mismo, y que, a su vez necesiten menos recursos computacionales. Debido a ello se plantearon las listas KBM2L, las cuales son estructuras que recogen todo la información de las bases de conocimiento y a su vez ocupan menos tamaño que sus contra-partes. Sin embargo esta aproximación tiene una desventaja y es que la búsqueda de la mejor KB2ML es un problema combinatorio complejo.

Se torna necesario entonces utilizar técnicas meta heurísticas como los algoritmos de estimación de distribuciones (EDAs), y que ayudan a buscar entre todas las posibles soluciones aquellas con las que obtendríamos una lista de menor tamaño mediante permutaciones de atributos y de dominio.

Por último, y debido a la demanda de las personas que toman las decisiones, se busca también que aquellos dictámenes que ofrezcan los software que toman decisiones aporten una explicación de porqué han tomado esa decisión y no otra, ya que permiten a las personas poder validar y entender el problema.

Abstract

The decision making process is a signature that has become more and more difficult these days and is no more than a reflection of the complexity of the world we live in and the speed at which it changes.

This is why decision support systems (DSS) were created a few decades ago. They provide a methodological and efficient framework for the analysis and evaluation of decision policies, thus saving people effort when making decisions.

However, even so, and due to the complexity of the knowledge bases which model decision problems, it is necessary to use structures that synthesise knowledge without altering it and that in turn require fewer computational resources. For this reason, KBM2L lists were proposed, structures that collect all the information of the knowledge bases and at the same time occupy less size than their counterparts. However, this approach has a disadvantage because the search for the best KB2ML is a complex combinatorial problem.

It then becomes necessary to use meta-heuristic techniques such as estimation distribution algorithms (EDAs), which help to search among all the possible solutions those with which we would obtain a list of smaller size through permutations of attributes and domain.

Finally, and due to the demand from decision-makers, it is also demanded that those opinions offered by software decision-makers provide an explanation of why they have taken that decision and not another, as it will allow people to validate and understand the problem.

Tabla de contenidos

Capítulo 1

Introducción

Este capítulo hace un breve preámbulo sobre el trabajo, su motivación y alcance. Además, se explican los objetivos del mismo y, por último, se hace un desglose de como esta distribuido la memoria en capítulos.

1.1. Definición, motivación y alcance

Vivimos en un mundo en el que cada vez se toman decisiones más complejas, las cuales tienen consecuencias muy grandes en numerosos ámbitos y que afectan cada vez a más personas. Pensemos en el caso de una empresa de tamaño grande, en el que el equipo de gobierno tiene que tomar una decisión que puede afectar el devenir de la empresa en los años venideros. Pensemos en los grupos de personas puede afectar esta decisión. Los llamados stakeholders son cada vez grupos más numerosos y variados, además de que ya no son solo los propios miembros de la empresa. Esto, junto a que tienen más fuerza en la toma de decisiones, y que, unido a un entorno muy cambiante, hace que alrededor del 65 % de las decisiones que se toman sean más complicadas que en la época pre-Covid19, según afirma Gartner [1].

La dificultad de las decisiones viene dada por varios factores, y es que, al contrario que hace unos años, ahora no se pueden tomar decisiones en base a hechos pasados, sino que se premia a la rapidez a la toma de decisiones (especialmente en un mundo tan cambiante como el nuestro), la gestión del riesgo, la consistencia, la escalabilidad, el entorno....

Es por ello que se necesitan cada vez más las llamadas bases de conocimientos y análisis de datos, que ayuden a los humanos a tomar decisiones, pudiendo escoger una metodología data-driven (basado en datos almacenados en bases de datos), model-driven (modelos basados en los requerimientos del cliente o usuario) o knowledge-driven (basados en estructuras de conocimiento), entre otras.

Para ello y dentro de la Teoría de la Decisión, se creo el concepto de los Sistemas de Soporte de Decisiones (o en ingles DSS), que empezó a ganar importancia en la década de los 70s. Estos sistemas nos permiten tomar decisiones difíciles en

un menor tiempo, reduciendo el coste bajo una gran incertidumbre. Además, esto se une al hecho de que cometer un error puede generar un gran coste, hay una gran cantidad de decisiones alternativas y donde no se puede basar las decisiones en el concepto de ensayo y error.

Con todo ello, estas bases de conocimiento pueden alcanzar un gran tamaño, debido a la complejidad del problema que trata (y de la intrínseca complejidad del mundo real), pudiendo exceder los recursos computacionales de la máquina donde trabaje este DSS. Debido a ello, se plantea el concepto de las listas KBM2L [2], que no es más que una forma de reordenar estas tablas o bases de conocimiento de tal forma que se minimiza la memoria que ocupan, manteniendo toda la información de la estructura original. Este proceso, que puede parecer a-priori sencillo, es un problema computacional realmente difícil, siendo de tipo NP-Completo, ya que debe de buscar, entre todas las posibles alternativas, la lista de mínimo tamaño que almacene todo el conocimiento original. Por consiguiente, se utilizan meta-heurísticas de optimización combinatoria y técnicas de computación de alto rendimiento para poder lidiar con este problema, que de otra manera se puede tornar imposible de resolver.

Por último, nos podemos plantear el motivo de porqué un DSS nos devuelve la decisión óptima X y no la Y, o bien las características principales a la hora de decidir tal cosa y no la otra. También podemos preguntarnos cómo entiende el modelo el problema. Estas son ideas que se enmarcan dentro de la explicabilidad y la interpretabilidad de los modelos de inteligencia artificial, no solo de los DSS, sino de otros modelos de inteligencia artificial que toman decisiones como los algoritmos de aprendizaje automático (ML) o aprendizaje profundo (DL). Por ello, en los últimos años, se ha vuelto popular el concepto de la inteligencia artificial explicable (XAI), que tiene como objetivo poder resolver estas preguntas y ayudar a cualquier persona que quiera aprender sobre ese modelo y aumentar la confianza de los usuarios en las decisiones que toma el modelo.

1.2. Objetivos

El siguiente trabajo aborda los temas mencionados, que se pueden sintetizar en los siguientes objetivos:

1. El concepto de los Sistemas de Soporte de Toma de Decisiones junto con sus representaciones matemáticas, aplicaciones en el mundo real y librerías asociadas.

2. Las técnicas de optimización, desarrolladas en R, para poder construir una KMB2L partiendo de una base de conocimiento, todo ello construido en un paquete en R.

3. Una forma de poder comunicar con el usuario las decisiones que toma el modelo y las razones por las que se basa esa decisión.

4. Pruebas del código realizado.

1.3. Descripción del documento

El presente trabajo se divide en varios apartados, siendo el primero el estado del arte, donde se explican las bases teóricas en las que se fundamenta el trabajo realizado. Se sigue por el apartado del desarrollo, en el que se describe el código desarrollado junto con una descripción del mismo. Luego, en el apartado de las resultados, se hace una comparativa de los códigos desarrollados y se mide el rendimiento obtenido. Posteriormente, se presentan las conclusiones del trabajo junto con las líneas futuras, y por último, se puede ver en el anexo el código realizado.

Capítulo 2

Estado del Arte

El siguiente capítulo describe los contenidos teóricos en los que se sostiene este trabajo. Se divide en tres secciones principales, siendo el primero los sistemas de soporte a decisiones, en el que se describe que son estos, sus representaciones matemáticas y sus aplicaciones y tendencias futuras. El segundo, la inteligencia artificial explicable, en el que se relata este concepto y la motivación detrás de este, junto con una breve explicación de las diversas técnicas de la misma. Por último, se habla sobre la optimización, que abarca distintas técnicas tanto matemáticas como de computación de alto rendimiento, que se han aplicado en la práctica.

2.1. Sistema de Soporte a Decisiones

Este apartado trata de dar un vistazo sobre los sistema de soporte a decisiones, una definición de los mismos y de su historia, sus características y representaciones, junto con sus aplicaciones y tendencias actuales.

2.1.1. Definición y Tipos

Los sistemas de soporte a decisiones (o por sus siglas en inglés DSS, Decision Support Systems) son una serie de programas que tienen como objetivo el facilitar la toma de decisiones. Esto lo hacen mediante el análisis de grandes cantidades de datos en forma de bases de conocimiento y tratan de inferir la decisión, al igual que lo haría un ser humano. Estos sistemas, que están dentro del dominio de la teoría de la decisión, modelan el problema presente en la vida real a un modelo matemático, con la ayuda o no de personas expertas en el problema que se quiere resolver. En resumen, un DSS puede construirse con datos y con expertos, su evaluación y análisis de los resultados genera una base de conocimiento que puede ser muy grande y de estructura compleja.

Estos programas tienen como origen los estudios llevados a cabo por la Universidad Carnegie Mellon (CMU) en la década de los 50s y los 60s del siglo XX. Esta universidad publicó una serie de implementaciones en los años 60s, pero no fue hasta la década siguiente y sobre todo en la década de los 80s hasta cuando se

vivió un boom de investigación en esa área. Algunas fechas interesantes son el año 1969, en el que se encuentra el primer uso de un DSS en un experimento científico. En 1971 se encuentra su primera denominación como tal en un articulo científico de Gorry y Scott Morton (los años 70 fueron muy prolíficos en cuanto al número de publicaciones sobre el tema [3]). Otra fecha importante fue 1974, fecha en la cual Gordon Davis, profesor de la Universidad de Minnesota publicó el famoso texto sobre los MIS (Management Information Systems), en el cual se dedica un capítulo a hablar sobre los DSS.

Los DSS cuentan típicamente con tres componentes, la base de conocimiento, el modelo y la interfaz con el usuario. En cuanto al primero, la base de conocimiento proviene de datos internos o externos y contiene información referente al dominio que trata el DSS y el cual este usa en su motor de razonamiento. El modelo permite realizar las decisiones conforme a las preferencias de los decisores. Además, permite comprender el problema, limitar su complejidad y riesgo. La interfaz de usuario permite a este último comunicarse con el DSS.

Dependiendo de la fuente de información primaria, los DSS se pueden dividir en cinco tipos, a continuación se da una breve descripción de estos:

- **Data-driven**: Usa técnicas de minería de datos para obtener tendencias o patrones y poder predecir el futuro. Es típico en este tipo de modelos tener acceso a series temporales o datos en tiempo real. Normalmente se usan bases de tipo "Data Warehouse" para almacenar los datos. Utilizados en decisiones relacionadas con inventarios, procesos de negocio, ventas...
- **Model-driven**: Usado en modelos financieros, de optimización o de simulación, usa datos limitados y con requisitos planteados por los decisores y expertos. En general, no es necesario grandes bases de datos para este tipo de modelos. Ejemplos de uso y comercialización de estos modelos son sistemas de planificación financiera.
- **Communications-driven**: En estos tipos de modelos se enfatiza la colaboración entre varios usuarios y el propio modelo mediante el uso de un conjunto de tecnologías de redes y de comunicación. Esto permite mejorar la eficiencia y la efectividad del sistema. Un ejemplo de uso puede ser en un equipo de desarrollo de una empresa, permitiendo poder realizar decisiones de manera conjunta y síncrona con todos los miembros del mismo.
- **Document-driven**: Usa tecnologías de análisis y de recuperación de textos permitiendo al usuario poder escanear documentos, páginas web o bases de datos, ayudando a encontrar términos o datos que ayudan a la toma de la decisión, como puede ser la búsqueda de catálogos o datos históricos de empresas. Su uso aumento desde la aparición y el uso común de internet.
- **Knowledge-driven**: En este tipo de modelos, los datos están almacenados en una base de conocimiento de un tema en concreto y actualizados gracias a los expertos del mismo tema. Sugiere acciones de manera inteligente a los usuarios y comprende el dominio del problema. En ellos se usan sistemas de bases de conocimiento. Ejemplos de uso pueden ser en el ámbito de la medicina.

Queda patente que sus usos son muy variados y que normalmente involucra personas con diferentes estudios y ámbitos, debido a la tendencia general de la dificultad a la hora de la toma de decisiones en todo tipo de áreas. Los DSS mejoran la eficiencia y la velocidad de la toma de decisiones, mitigando el coste y el riesgo además de permitir al usuario aprender sobre el dominio del problema. Además, son adaptativos con el cambio del tiempo, especialmente útil en un mundo como el nuestro, en el que predomina la incertidumbre. También y gracias a los avances en la potencia de los ordenadores, los DSS pueden analizar cada decisión de manera muy rápida.

Cabe mencionar también que existen dos tipos de decisiones posibles [4], siendo la primera las decisiones programadas, y la segunda, las no programadas. Las programadas son aquellas que se repiten durante el tiempo (ya se han tomado esas mismas decisiones previamente) y las cuales siguen una serie de reglas que se pueden generar. Un ejemplo de estas puede ser el inventariado de un almacén, que dependiendo de unas reglas previas, se hará de una forma u otra.

Las no programadas son aquellas, que como el nombre indica, se basan en criterios no muy bien definidos y que ocurren de manera inusual. Por consiguiente, la información que recibe el decisor puede ser ambigua o incompleta, siendo imprescindible en este caso el buen juicio y la experiencia de la persona o grupo de personas que tome la decisión. Un ejemplo típico de este tipo de decisiones puede un inversor que tiene la decisión de invertir o no en una nueva tecnología. Los DSS pueden realizar ambos tipos de decisiones, siendo más frecuente que estos trabajen en las decisiones no programadas.

Por último, el proceso que se lleva a cabo a la hora de tomar una decisión es el mismo que realizan tanto los DSS como los seres humanos. Se puede ver un ejemplo de esto en la imagen 2.1. De forma resumida, primeramente, se identifica que decisión tenemos que tomar y pensar en sus múltiples alternativas. Luego tenemos que analizar cada alternativa individualmente y escoger la mejor de ellas según una serie de criterios (más adelante se introducirá el concepto de la utilidad) que pueden ser definidos por una serie de expertos. Por último, se implementará y evaluará la opción escogida.

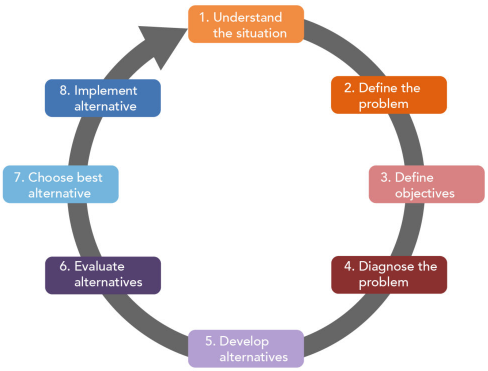


Figura 2.1: Ciclo toma de decisiones

2.1.2. Representaciones

Se ha mencionado que una parte esencial de los DSS es el modelo. Para este trabajo se utilizaran los llamados diagramas de influencia, que a su vez y gracias a estos, se generarán tablas de decisiones óptimas, que contendrán las reglas del dominio del problema y su utilidad. También se presentarán los llamados árboles de decisión. Todas estas representaciones matemáticas son necesarias debido a que modelan un problema real de tal forma que es entendible para las máquinas y los seres humanos [5].

2.1.2.1. Diagramas de Influencia y concepto de Utilidad

Los diagramas de influencia (ID en ingles, Influence Diagrams) son grafos acíclicos (es decir, que no tienen ciclos) que tratan de modelar un problema real de decisión que tiene incertidumbre. Planteados por Howard y Matheson en 1984 [6], estos codifican los tres elementos básicos de una decisión: las distintas soluciones disponibles, factores relevantes a la decisión (como por ejemplo, como se relaciona e interactúa con otras decisiones) y las preferencias del propio decisor o experto.

Para ello, los ID usan tres tipos de nodos, los decisores, las variables aleatorias y los objetivos (o nodos valor). El nodo decisor, que normalmente están representados como rectángulos, contiene la decisión a tomar (como por ejemplo, el invertir o no en un producto) y que puede ser modificado directamente por el experto. Las variables aleatorias, representadas como óvalos, manejan incertidumbre e información incompleta y no pueden ser modificadas directamente aunque podemos considerar instancias en algunas variables para que se propaguen como evidencia.

Nótese que, tanto las variables aleatorias como los nodos decisores, en el caso de este trabajo, son discretos aunque también pueden tomar valores continuos. Por ejemplo, en el caso del nodo decisor,la decisión puede ser binaria (invertir o no invertir, siguiendo el ejemplo previo) o puede tomar más de dos valores. El nodo valor, objetivo o utilidad es una medida de satisfacción con respecto a los distintos resultados, es decir, cuantifica la preferencia. Normalmente esta representado con figuras hexagonales o en forma de diamante. Por último, cada nodo esta unido mediante una flecha. Esta flecha indica influencia sobre un nodo a otro (y con los nodos previos a su vez) además de una relación temporal. Esta relación puede o no implicar causalidad.

Siguiendo el ejemplo de la figura 2.2, vemos que hay un nodo decisor (D), cuya decisión es fumar o no fumar. También se puede observar que hay un nodo valor llamado Value (D), que esta en función de Lung Cancer (C) y Smoking (S), y que expresa las preferencias del experto de las distintas posibilidades del proceso de decisión. En el caso del nodo E, vemos que se sabe su estado antes de que se tome la decisión (D), ya que hay una flecha del primero al último (denotando la relación temporal).

La evaluación de un modelo de decisión consiste en calcular todas las utilidades para todas y cada una de las posibles combinaciones de decisiones. Cabe

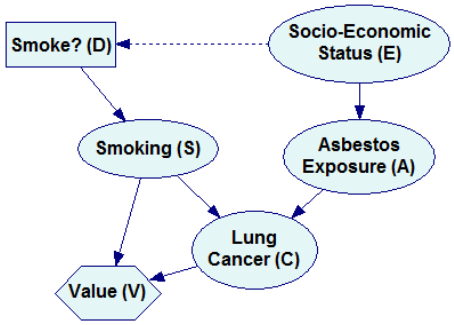


Figura 2.2: Ejemplo Diagrama de Influencia de GeNIe

destacar que puede haber mas de un nodo decisor siempre y cuando el flujo del diagrama acabe en un mismo nodo valor. Las utilidades obtenidas y con las que se escogerá una decisión o no, son condicionadas sobre los distintos valores que pueden tomar los nodos aleatorios.

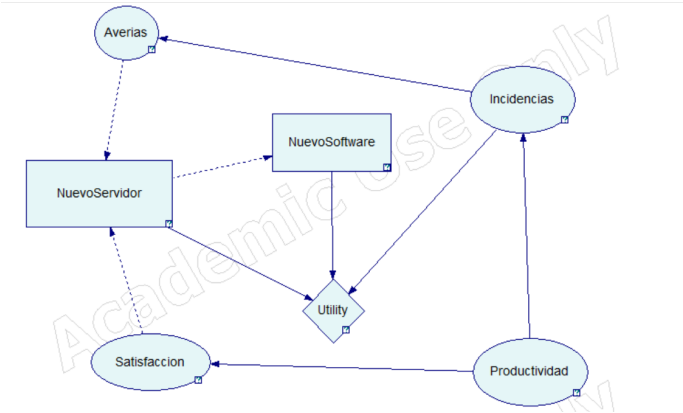


Figura 2.3: Diagrama de Influencia con 2 Nodos Decisores de GeNIe

En la figura 2.3, se puede ver que hay dos nodos decisores, NuevoServidor y NuevoSoftware y que se conoce la primera decisión antes de tomar la segunda (vemos que el flujo del diagrama siempre lleva al nodo utilidad). La decisión óptima no es mas que aquella que tenga una mayor utilidad, dentro de los posibles valores que tengan los nodos aleatorios. Nótese que el calculo de la utilidad se simplifica si se conoce los valores que toman los nodos aleatorios.

El cálculo de la utilidad se realiza mediante el Teorema de Bayes y y el principio de Máxima Utilidad Esperada. A posteriori, el experto(s) valida esos valores, además de proponer y ayudar en el diseño las propias variables y el grafo (que relaciones hay, las relaciones prohibidas...). Una vez que tengamos el resultado, el paso a las tablas de decisión es directo, como se explicará en el apartado 2.1.2.3.

El concepto de utilidad, mencionado durante esta sección, es introducido en la teoría de la decisión como un elemento que ayuda a seleccionar una sentencia frente a otra. Esta utilidad sirve para mapear las posibles decisiones alternativas

con un conjunto de números reales (que toman valores continuos, sin ningún valor supremo ni un valor 0). Cabe destacar que la utilidad es subjetiva y depende de los constructores del problema de decisión (proceso conocido como obtención de la utilidad). En resumen, la utilidad modela las preferencia del decisor sobre una escala numérica ordinal.

Por último, hay que hablar de las diferencias entre las Redes Bayesianas (RRBB) y los diagramas de influencia. Y es que los diagramas de influencia son una extensión de las redes Bayesianas y cuya diferencia es que en los primeros hay una declaración explicita de diferentes decisiones y preferencias en los resultados de la decisión. En los formalismos matemáticos de las redes Bayesianas, los arcos no implican causalidad mientras que en las diagramas de influencia si ocurre (siempre que sean originados de un nodo de decisión).

2.1.2.2. Árboles de Decisión

Los árboles de decisión (no confundir con el método de aprendizaje automático de tipo supervisado que tiene el mismo nombre) son un modelo de tipo árbol jerárquico y que muestra visualmente las decisiones y sus potenciales consecuencias y costes, es decir, muestra el proceso de decisión de manera gráfica. Cada rama se puede considerar como una decisión y gracias a ello se pueden comparar para buscar cual es la mejor de ellas. Es una representación cronológica del proceso de decisión.

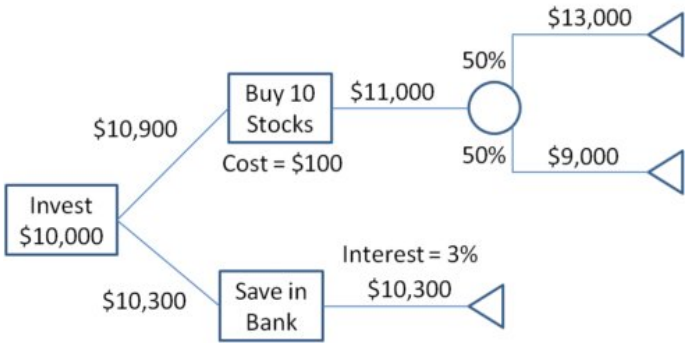


Figura 2.4: Ejemplo Árbol de Decisión [7]

Al igual que en cualquier modelo de tipo árbol, existen los nodos raíz y hoja, siendo el nodo raíz el nodo que representa el objetivo o decisión última a tratar. En el caso de la figura 2.4, vemos que la decisión es la de invertir 10000 dolares. Siguiendo el ejemplo, cada bifurcación o alternativa indica el curso de la acción a seguir o tomar, en este caso se presenta la decisión de guardar el dinero en el banco o comprar 10 acciones (con un coste asociado). Cada nodo hoja representa los resultados de la acción y pueden ser representados normalmente mediante un círculo o un cuadrado, según su tipo. El primero indica las decisiones a tomar mientras que los segundos denotan nodos probabilísticos, cuyo resultado es desconocido. Por último, los nodos terminales, representados mediante un triángulo, son el final de la decisión en conjunto y están las utilidades de cada escenario o camino del árbol.

Una vez que tenemos el modelo planteado, con sus distintas alternativas, se puede utilizar técnicas típicas de la investigación operativa, como la poda, para poder obtener la decisión óptima. Cada nodo hoja de tipo decisión viene asociado con una función de tipo *"payoff"*, que no es más que las acciones o reacciones, mientras que los nodos probabilísticos tienen asociados una función de probabilidad. Esta función se puede modelizar como una función de utilidad. Con todo ello, un árbol de decisión, denotado como DTs(Decision Trees), se puede modelizar como un conjunto de nodos, vértices, funciones de payoff y de probabilidad.

Los arboles de decisión pueden ser construidos de manera manual, aunque se pueden construir también por software, como por ejemplo, mediante Microsoft Excel usando el plug-in Tree-Plan [8]. Los árboles se leen de izquierda a derecha. Este modelo además denota relación causal y temporal (cada nivel del árbol denota diferentes pasos en el tiempo). Una vez construido el árbol, este se puede leer mediante sentencias condicionales del tipo "Si x es ... entonces y, de otra manera entonces z".

Este tipo de modelos tienen varias ventajas. La primera y la más evidente es la sencillez de su entendimiento, especialmente para las personas no expertas (al igual que su contra-parte en los modelos de aprendizaje automático) junto a que modelizan problemas complejos de manera efectiva. Además, estos son muy flexibles, pudiendo añadir nuevas decisiones de manera sencilla e intuitiva. Por último, identifican rápidamente las ganancias y las perdidas de cada decisión, los objetivos y las probabilidades. En cuanto a las desventajas, la más evidente es la complejidad del calculo, al igual que en un problema de heurística, en un modelo grande, el tiempo de cálculo de la decisión óptima puede escalar rápidamente. Otra desventaja es la inestabilidad del modelo, esto es, que un cambio en un nodo puede afectar sustancialmente a los nodos hijos y a los suyos a la vez.

El diagrama de influencia es muy compacto, pero su uso es para problemas simétricos y sin restricciones, mientras que los árboles de decisión (y las tablas de decisión 2.1.2.3) son combinatorios, complejos, pero permiten coalescencia y podas por asimetrías intrínsecas del problema. Una asimetría se debe a que una decisión o variable de azar depende de cierta información no disponible porque una decisión contempla que no esté disponible, como por ejemplo, que si vendemos una casa no podemos ya reformarla o alquilarla (decisión que restringe alternativas en decisiones posteriores).

2.1.2.3. Tablas de Decisión y Tablas de Decisiones Óptimas

Las tablas de decisión son un tipo de representación estructurada en el que cada fila de la tabla consiste en una serie de condiciones lógicas y cada columna una serie de acciones u condiciones a considerar. Se pueden entender como una estructura *if-then-else* en el que es posible que exista más de una acción. Si se cumplen todas las condiciones, se ejecutaría esa regla o política. Es una forma intuitiva y fácil de representar, en el ámbito de la teoría de la decisión, todas las acciones o decisiones en el problema a tratar. Se puede encontrar de varias maneras la representación de esta estructura, siendo lo más común que

las reglas se lean de izquierda a derecha en la tabla aunque también se puede encontrar que se puedan leer de arriba a abajo.

		Rules							
Conditions	Printer prints	No	No	No	No	Yes	Yes	Yes	Yes
	A red light is flashing	Yes	Yes	No	No	Yes	Yes	No	No
	Printer is recognized by computer	No	Yes	No	Yes	No	Yes	No	Yes
Actions	Check the power cable			✓					—
	Check the printer-computer cable	✓		✓					—
	Ensure printer software is installed	✓		✓		✓		✓	—
	Check/replace ink	✓	✓				✓		—
	Check for paper jam		✓		✓				—

Figura 2.5: Ejemplo Tabla Vertical

En el ejemplo de la figura 2.6, vemos que las condiciones pueden ser binarias o no, siempre categóricas. Las propias columnas pueden ser también acciones a tomar e indicarían temporalidad en la secuencia de la propia decisión. La última columna es la decisión final o el objetivo del problema (en el caso de la figura, el hecho o no de operar). En la figura 2.5, que muestra las decisiones a seguir si se estropea la impresora, se indica las acciones que se deben hacer o no según si se cumplen las reglas de manera vertical. Por ejemplo, si la impresora no imprime y una luz roja parpadea y la impresora no es reconocida por el ordenador, cumpliría las condiciones de la primera regla y se harían esas acciones en ese orden.

Las tablas pueden incluir o no todas las posibles combinaciones del dominio, en caso afirmativo, se considera que la tabla esta balanceada o completa. Visualmente son muy poderosas y fáciles de entender pero en problemas complejos pueden ocupar mucha memoria.

PAIN	ANGIOGRAM	SURGERY	Utility
ABSENT	NEGATIVE	→NO	**0.74673**
ABSENT	NEGATIVE	YES	0.64070
ABSENT	POSITIVE	→NO	**0.65233**
ABSENT	POSITIVE	YES	0.64598
PRESENT	NEGATIVE	→NO	**0.74453**
PRESENT	NEGATIVE	YES	0.64083
PRESENT	POSITIVE	NO	0.63965
PRESENT	POSITIVE	→YES	**0.64668**

Figura 2.6: Ejemplo Tabla de Decisión Óptima

Como se mencionó en el punto 2.1.2.1,los diagramas de influencia se pueden convertir a tablas de decisión. Una vez evaluado el diagrama de influencia, la tabla generada se considera una tabla de decisiones óptima, en el que tenemos una columna de utilidades, todo ello basado en la teoría de juegos planteado por Neumann et al. en su libro sobre la teoría de juegos [9]. Un ejemplo de tabla de decisiones óptima es la de la figura 2.6. El proceso es análogo con los árboles de decisiones, descritos en el punto 2.1.2.2.

Rest1	R11								
Rain1	None1			Low1			High1		
Rest2	R21	R22	R23	R21	R22	R23	R21	R22	R23
Exp. utility	0.48971045	**0.78838...**	0.51394203	0.60975327	0.84194055	0.37032117	0.60975327	0.84194055	0.37032117

Figura 2.7: Evaluación diagrama de influencia en el interfaz de GeNIe

La tabla de decisión es un modelo equivalente al árbol, al diagrama de influencia y a la tabla de decisión, y consiste en columnas con el estado (aleatorio) del problema (producto cartesiano de los dominios de las variables y la distribución conjunta), y por filas el producto cartesiano de las alternativas de las decisiones. Es la RAW Knowledge Base, es decir, punto de partida para desarrollar una base de conocimiento eficiente (almacenamiento y acceso) e interpretable (explicaciones). Sencilla en casos donde solo haya una o dos variables y un solo punto de decisión pero el tamaño crece según aumente la complejidad del problema, ocasionando que los requisitos computacionales crezcan. El caso de la figura 2.7 es una tabla de decisiones óptimas que se ha obtenido tras evaluar el diagrama de influencia mediante el software GeNIe [10]. Nótese que es una representación extensiva.

2.1.3. Aplicaciones y Tendencias

Desde 2003 existe la asociación llamada *International Society for Decision Support Systems* (ISDSS), cuya misión es la de divulgar, investigar y ofrecer ayuda a empresas sobre los DSS. Los sistema de soporte de decisiones continuarán expandiéndose a nuevos mercados y aprovecharán las nuevas técnicas de inteligencia artificial y almacenamiento de datos junto con las mejoras de las prestaciones de los ordenadores. Se podrán modelizar problemas más complejos y las nuevas tecnologías de comunicación supondrán una mejora sustancial para los DSS de tipo comunications-driven.

En cuanto a las aplicaciones, los DSS no se limitan a un ámbito en concreto, sino que se encuentran ejemplos en un gran número de disciplinas. Desde los GPS, pasando por la agricultura, hasta la medicina. Un ejemplo de este último, es el modelo de tratamiento de ictericia neonatal que desarrollo el departamento de inteligencia artificial (DIA) de la Universidad Politécnica de Madrid (UPM) [11]. Otro ejemplo comúnmente usado son en dashboards empresariales (ERP).

2.2. Inteligencia Artificial Explicable

En el siguiente apartado se trata el concepto de la inteligencia artificial explicable, su origen, motivación y objetivos. También se presentarán diversas técnicas de la misma, como el método SHAP [12] y permutation importance [13].

2.2.1. Descripción y Motivación

Se ha mencionado en los capítulos previos, la gran variedad de uso de los DSS (al igual que la Inteligencia Artificial en general) además de que estos toman

decisiones difíciles, con gran incertidumbre e impacto. Una de las grandes desventajas de los DSS, al igual que algunos modelos de IA, es que son opacos y que solo dan un output en base al input que le damos, sin ofrecer explicaciones del por qué de la decisión. Esto puede ocasionar recelos o desconfianza, especialmente en aquellos casos que sean decisiones de alto riesgo, como en el caso de un diagnostico médico.

Es por ello que nace el concepto de inteligencia artificial explicable (en ingles XAI) [14], cuyo objetivo no es más que proponer que los algoritmos sean explicables, esto es, que su funcionamiento sea fácil y sencillo de entender. En el caso de los DSS (o cualquier modelo de IA que tome decisiones) esto se consigue mediante el entendimiento de las decisiones (a posteriori) que tome. A consecuencia de esto, nos puede ayudar a entender mejor el problema que estemos tratando.

Este conjunto de técnicas o métodos se fueron desarrollando paralelamente a lo largo tiempo, de la misma forma que los algoritmos de inteligencia artificial. Con el surgimiento de los métodos de aprendizaje profundo (extremadamente opacos), han aparecido nuevas técnicas que buscan hacer estos algoritmos mas transparentes para el usuario. Además, entidades reguladoras, como la Unión Europea, han legislado en favor de la explicabilidad y la transparencia de los algoritmos. Un ejemplo de ello es el Reglamento General de Protección de Datos (GDPR).

No hay que confundir, en el dominio de la XAI, el concepto de explicabilidad e interpretabilidad, ya que hacen referencia a definiciones distintas. En cuanto a la interpretabilidad, esta se refiriere a como los humanos entendemos el comportamiento interno del modelo (sus operaciones internas,el proceso por el que da el output). Normalmente, un modelo es más interpretable que otro si tenemos más facilidad de comprender los pasos internos que toma el modelo para llegar a al predicción. También a mayor interpretabilidad, menor es la necesidad de usar técnicas externas que nos ayuden a entender el modelo.

Sin embargo, a mayor interpretabilidad, menor es la capacidad o potencia de ese modelo. Ejemplos de modelos interpretables son las regresiones lineales o los arboles de decisión (no confundir con los del apartado 2.1.2.2) mientras que las redes de neuronas son un caso de lo opuesto [15]. Un modelo como los arboles de decisión son interpretables porque son modelos visuales, cuya traza es fácil de seguir, al igual que las regresiones lineares, ya que el output está en función de una ecuación lineal que usa los parámetros de entrada.

Mientras que la explicabilidad hace referencia a como las mecánicas internas del modelo afectan al resultado. Se centra en el comportamiento del modelo, de darle visibilidad y de no tratar el modelo como si fuese una caja negra. Se usan técnicas externas que nos ayuden a comprender los motivos que llevan al modelo a tomar una decisión en concreto. Estas técnicas se usan por ejemplo, en modelos como los Random Forest (métodos de tipo ensemble, formados por varios arboles de decisión), que por si solos son muy difíciles de entender. Con estas técnicas podemos discernir porqué hace tales decisiones con determinados inputs o que inputs impactan más en los outputs. Esto último es especialmente útil en el trabajo realizado.

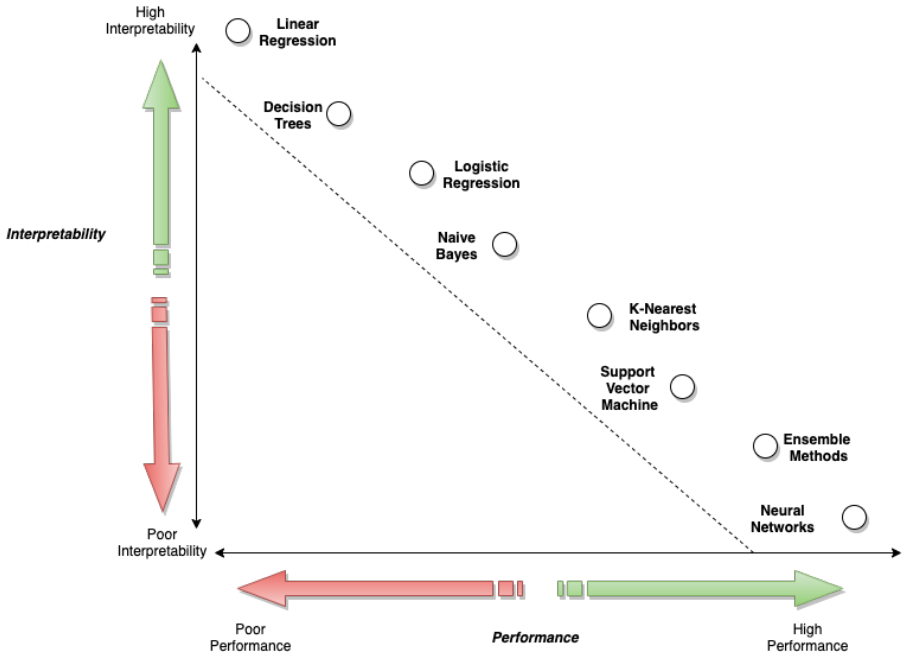


Figura 2.8: Interpretabilidad vs Rendimiento [16]

Es importante notar que esta capa de explicabilidad que se le dan a los modelos es posterior a la toma de decisión, simplemente se le esta dando claridad al modelo. Con ello, puede permitir a las partes implicadas aprender sobre el problema y verificar el resultado o incluso ajustar el modelo. Por último, hay que mencionar el tema de la ética [17], que aunque no es un objetivo del trabajo, es un área importante, no solo de los DSS sino de la IA en general. Preguntas como, quien tiene responsabilidad sobre las decisiones del modelo o la falsa confianza que nos da la explicabilidad del modelo, no pueden ser olvidadas.

2.2.2. Técnicas de Explicabilidad

Debido a la gran cantidad de técnicas [16], en este trabajo se presentan solo dos de ellas, el método SHAP y permutation importance. Hay otros métodos como los partial plots o los metamodelos, que deben ser conocidos por cualquier persona que quiera trabajar con modelos de IA en general. En el caso de permutation importance, esta técnica intenta determinar como cada atributo o característica afecta a la predicción mientras que los métodos SHAP averiguan como contribuye cada variable de manera cuantitativa a la predicción. Es importante decir que estás técnicas dan aproximaciones y que se deben apoyar con otras estrategias como la visualización de datos u otras técnicas para poder dar una visión más completa y fiable.

2.2.2.1. Método SHAP

Los valores SHAP (acrónimo de SHapley Additive exPlanations) es un método basado en la teoría de juego [9] que tiene como objetivo explicar los outputs de

cualquier modelo de IA [12]. Esto lo hace mostrando la contribución de cada atributo o variable al output recibido (importante, no esta validando el modelo en el cual se esta aplicando).

Si tenemos un conjunto de atributos y un conjunto de muestras, los valores SHAP descomponen la contribución de cada atributo al output comparando a su vez con el resto de atributos del conjunto. Si se cambiase el conjunto de muestras, no se garantizaría que se obtuviese el mismo resultado.

Para explicar aun mejor el concepto se plantea el siguiente ejemplo: estamos desarrollando un modelo que predice si un paciente necesita estar hospitalizado o no y tenemos un conjunto de atributos que representan el estado de un paciente al llegar al hospital. Sabemos que el modelo ha predicho que el paciente necesita hospitalización, entonces, nos podemos preguntar cómo actuaria los valores SHAP. Este cogería el atributo nivel de fiebre, que en el caso del paciente es alto y con un conjunto de instancias, los valores SHAP compararían ese valor con el resto de atributos del paciente y ven el impacto sobre el output dado.

Una vez realizado todas las operaciones, se obtiene una serie de valores por cada atributo, que pueden ser tanto positivos o negativos, que indican su influencia (tanto positiva como negativa) al output dado. Un atributo a tenido mayor influencia que otro si el valor absoluto del SHAP es mayor que el otro. En la imagen 2.9, se puede ver un ejemplo del resultado del valor SHAP.

Figura 2.9: Ejemplo Valores SHAP

La imagen 2.9, que representa los valores SHAP en los atributos (o en este caso, las estadísticas) de un equipo de fútbol a la hora de predecir que ha ganado un partido. La confianza de la predicción es de 0.7, mostrando en rojo los atributos (por orden de impacto) que han contribuido a que el equipo haya ganado (con ese valor 0.7 que ha predicho el modelo). Lo que mas ha contribuido al modelo a dar ese resultado es que el equipo ha marcado 2 goles en el partido, mientras que el atributo que más ha influenciado en el lado contrario (azul) es la posesión del 38 % del equipo (lo cual tiene sentido en ambos casos).

Los valores SHAP están implementados en el lenguaje Python en la librearía SHAP y pueden ser usados en cualquier modelo. Por último, los valores SHAP nos indican la importancia de cada característica y nos ayuda a entender el modelo e incluso mejorándolo.

2.2.2.2. Permutation Importance

La técnica de permutation importance [13] trata de buscar que atributos o características afectan más el output. Lo hace mediante la permutación de las mismas, y es que si, al cambiar de sitio esos valores (y por ende, rompiendo la

relación entre esa misma variable y el output) se reduce la fiabilidad del modelo (precisión, MAE, r-square, etc), entonces es un indicativo de la importancia de esa variable.

La ventaja de la técnica radica en que no se necesita volver a entrenar el modelo para poder aplicarla, ya que se hace después de que el modelo este entrenado. Simplemente se mezclan los valores de una columna o característica de forma aleatoria y se evalúan los cambios en la fiabilidad del modelo. Un efecto secundario de este método es que también rompe las relaciones con los otros atributos, lo cual es un indicativo de que tiene en cuenta todas las interacciones con otras variables. Por último, es independiente del modelo, ya solo observa las variaciones del output con respecto de la variación del mismo.

Sin embargo, la buena evaluación de la permutación radica en que el propio mezclado del atributo sea bueno, ya que los errores pueden variar más o menos según el mezclado (pudiendo dar interpretaciones erróneas). Además es sensible a atributos que tengan una correlación alta, ya que puede dar lugar a instancias de variables irreales y incrementar la dificultad de la interpretabilidad del resultado.

Weight	Feature
0.1750 ± 0.0848	Goal Scored
0.0500 ± 0.0637	Distance Covered (Kms)
0.0437 ± 0.0637	Yellow Card
0.0187 ± 0.0500	Off-Target
0.0187 ± 0.0637	Free Kicks
0.0187 ± 0.0637	Fouls Committed
0.0125 ± 0.0637	Pass Accuracy %
0.0125 ± 0.0306	Blocked
0.0063 ± 0.0612	Saves
0.0063 ± 0.0250	Ball Possession %
0 ± 0.0000	Red
0 ± 0.0000	Yellow & Red
0.0000 ± 0.0559	On-Target
-0.0063 ± 0.0729	Offsides
-0.0063 ± 0.0919	Corners
-0.0063 ± 0.0250	Goals in PSO
-0.0187 ± 0.0306	Attempts
-0.0500 ± 0.0637	Passes

Figura 2.10: Ejemplo Feature Importance usando ELI5

La figura 2.10 muestra un ejemplo del uso del método usando la librería ELI5 de Python. Esta librería permite visualizar y depurar los modelos de aprendizaje automático y permite utilizar técnicas para dar explicaciones de los mismos. Siguiendo el ejemplo del apartado 2.2.2.1, la imagen muestra la importancia de cada característica (o estadística del equipo en el partido) y las ordena de mayor o menor peso (+/- un intervalo, que depende de como haya sido la permutación). Como cabría esperar, la variable que tiene más peso son los goles (ya que dependiendo del número de goles que se marcan, puede hacer que el equipo gane o no). También se ve en la figura que hay variables con un valor negativo indicando que la variable no tiene impacto en el resultado, pero que, al modificar los atributos, ha resultado en que el modelo se comporta mejor (los atributos que no tienen importancia suelen tener un valor cercano al 0).

2.3. Optimización

En este apartado se enumeran y describen los métodos de optimización empleados en este trabajo, siguiendo el segundo objetivo de la memoria. De mencionan por un lado los algoritmos EDAs y TAN y por otro lado, las técnicas de computación de alto rendimiento.

2.3.1. Algoritmos de Estimación de Distribuciones (EDAs)

En esta sección se describen las técnicas meta-heurísticas de optimización usadas en el trabajo, en concreto, se describen un conjunto de algoritmos llamados EDAs y se desarrolla su funcionamiento y sus tipos y los métodos concretos que se han usado en el trabajo.

2.3.1.1. Definición general de los EDAs

Los EDAs (Estimation Distribution Algorithms) son un conjunto de algoritmos, que se enmarcan dentro de la computación evolutiva, que trabajan con un conjunto de datos iniciales, también denominado población y en los que se busca evaluar esa población con una función objetivo llamada "fitness". Una nueva población es generada en base a la función objetivo y el proceso se repite iterativamente hasta que el óptimo se haya encontrado o se haya cumplido la condición de finalización del algoritmo.

Planteados en 1996 por Muhlenbein y Paaß [18] como contra-parte de los *Algoritmos Genéticos (GE)* [19], estos primeros no tienen operadores de cruce y mutación además de las probabilidades de cruce y mutación asociadas, usadas en la creación de la nueva población. Esto es una ventaja, ya que la búsqueda de unos buenos parámetros en los algoritmos genéticos es esencial para el buen rendimiento de los mismos y se puede convertir a su vez en un problema de optimización.

En el caso de los EDAs, la nueva población de individuos se obtienen mediante la simulación de una distribución de probabilidad, calculada gracias al conjunto de individuos obtenidos en la iteración o generación anterior, mientras que en el caso de los GA, la nueva población se obtiene mediante los operadores de cruce y mutación. Esto permite a los EDAS tener mas transparencia y expresividad frente a los GA, pero la estimación de la distribución de probabilidad conjunta asociada a los individuos seleccionados en cada iteración se vuelve el cuello de botella de esta aproximación. Los GA sin embargo generan la nueva población mediante la recombinación de las soluciones prometedoras.

En general, los EDAS han mostrado ser mejores en algunos aspectos frente a los GA, siendo algoritmos robustos en la mayoría de situaciones. Su uso es amplio, desde áreas como la bioinformática [20], en el aprendizaje automático o en problemas combinatorios. En el caso de este trabajo (optimización de bases de conocimiento), se han usado ambas aproximaciones, resultando mejor el uso de los EDAS cuando el tamaño del problema crece.

2.3.1.2. Descripción del algoritmo

Los EDAs son un conjunto de algoritmos evolutivos que trabajan con una población de posibles candidatos a ser la solución óptima. En la imagen 2.11 se puede ver el diagrama de flujo general de los EDAs. En la fase inicial, se genera de manera aleatoria un conjunto de muestras, que será evaluada con una función objetivo o *fitness*. Esta función objetivo mide como de precisa es la muestra evaluada con la meta que se quiere conseguir. Luego, basado en los resultados que se obtienen al evaluar población inicial, se escogen un subconjunto de elementos de la muestra inicial, que tienen mejores valores en la función objetivo.

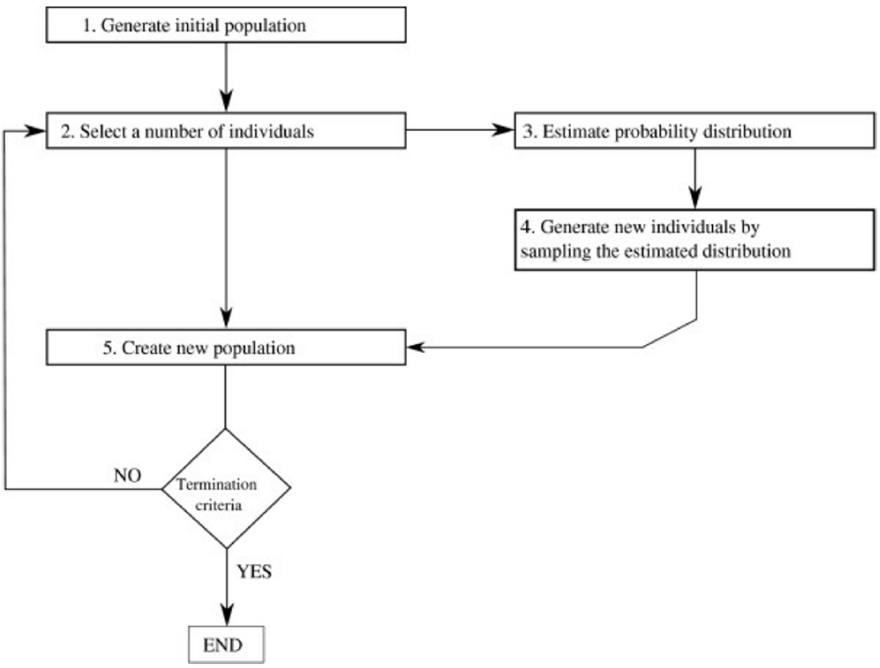


Figura 2.11: Diagrama de flujo general de los EDAs

A continuación, se construye un modelo probabilístico, que será diferente dependiendo del tipo de algoritmo que estemos usando y se generan nuevas muestras del modelo. Estas nuevas muestras tienen mejores resultados en la función objetivo y así, sucesivamente, se itera hasta que en la nueva muestra se encuentre presente el óptimo o hasta que se cumpla la condición de la función objetivo, haciendo que no se necesite operadores de mutación y cruce (que usan los GA).

Conceptualmente, el objetivo de los EDAs es buscar de todo el espacio de posibles soluciones aquellas áreas más prometedoras y guiar al algoritmo por esas áreas hasta encontrar el óptimo. Además, los EDAs son capaces de capturar diferentes patrones de interacciones entre los subconjuntos de las variables del problema y así generar nuevas poblaciones. Esto es debido al uso de las distribuciones de probabilidad conjunta (simplificada y/o factorizada), que construyen el modelo gráfico probabilístico, asociadas a los individuos seleccionados en cada iteración.

Siguiendo la figura 2.12, los N individuos se escogen usando los diversos méto-

EDA
$D_0 \leftarrow$ Generar M individuos al azar

Repetir for $l = 1, 2, \ldots$ hasta
que se cumpla el criterio de parada

$D_{l-1}^{Se} \leftarrow$ Seleccionar $N \leq M$ individuos
de D_{l-1} acorde con el método de selección

$p_l(\boldsymbol{x}) = p(\boldsymbol{x}|D_{l-1}^{Se}) \leftarrow$ Estimar la
distribución de probabilidad
de los individuos seleccionados

$D_l \leftarrow$ Muestrear M individuos (la nueva
población) a partir de $p_l(\boldsymbol{x})$

Figura 2.12: Pseudocódigo de un EDA [21]

dos que existen en computación evolutiva, como por ejemplo, la truncación y los subconjuntos que se toman pueden ser de diversos tamaños siempre que sean menores que el conjunto inicial. En caso de empates en la función objetivo, se pueden aplicar diversos métodos de selección, como la distribución uniforme. Una vez calculada la distribución de probabilidad se muestrean M nuevos elementos o individuos y se vuelve a iterar. Finalmente en la figura 2.13 se expresa todo lo explicado de forma visual.

2.3.1.3. Tipos de EDAs y sus características

Existen una gran variedad de tipos de EDAs y su uso depende del problema al que nos estemos enfrentando. Se tiene que tener en cuenta características del algoritmo tales como la complejidad del modelo probabilístico o el coste computacional en términos de CPU (Unidad Central de Procesamiento) y memoria de aprender el modelo y almacenar la población. Todo esto esta relacionado con el tipo de problema que estamos tratando de resolver, como el número de variables que tiene o el tipo de variables de las mismas (discretas o continuas).

Otra consideración es que cuando más simple sea el modelo probabilístico menor sera los requisitos de computación pero menor será su capacidad para modelizar problemas complejos (convergencia lenta). A veces, según el tipo de problema, se tendrá que optar por encontrar soluciones óptimas relativas ya que a mayor complejidad del problema, se necesitará un modelo más potente y los requisitos asociados en términos de rendimiento, estructuras de memoria, etc aumentarán. Por último, hay que tener en mente también si tenemos un conocimiento previo del modelo, por ejemplo, que variables son más importantes o cuales no, ya puede ayudar a escoger el modelo probabilístico.

Los EDAs pueden ser continuos o discretos y se dividen en tres categorías según como captan las interdependencias de las variables del problema, siendo estas las univariables, bivariables o multivariables (cabe resaltar que también hay

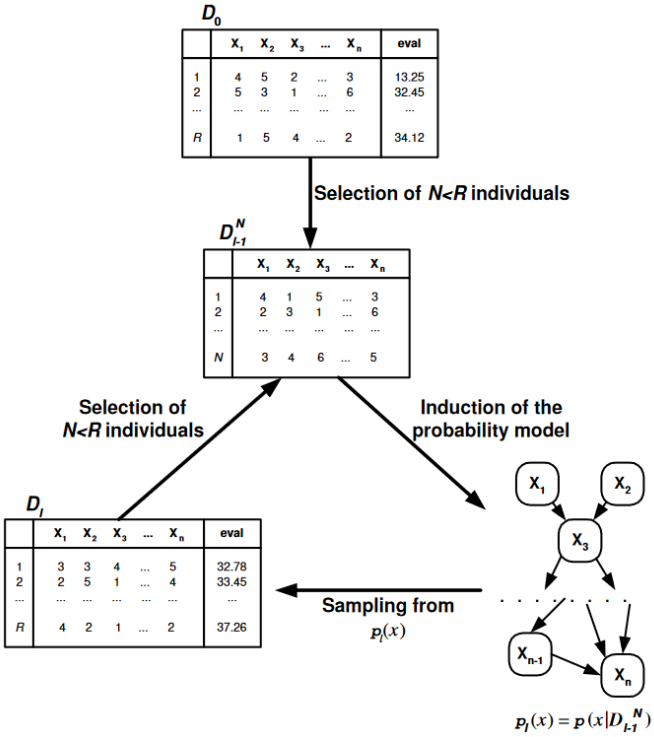


Figura 2.13: Explicación pasos de un EDA [20]

modelos mixtos):

- **Univariables**: también denominados como "EDAs sin dependencias", este tipo de algoritmos asumen que todas las variables son independientes y factorizan la probabilidad conjunta de los puntos seleccionados como el producto de la distribución marginal univariable. Por consiguiente, este subconjunto de EDAs son muy sencillos y rápidos pero la hipótesis de independencia está alejada de lo que puede ocurrir en un problema de optimización combinatoria, en el cual existen interdependencias entre las variables y funcionan mal en problemas donde es común llegar a óptimos locales ("deceptive problems"). Son aplicados en problemas de representación continua. Ejemplos de EDAs de este tipo son el UMDA, PBIL (Population Base Incremental Learning) o cGA (compact Genetic Algorithm).

- **Bivariables**: también denominados como "EDAs con dependencias bivariadas", son EDAs que son capaces de representar dependencias de orden inferior entre variables y ser aprendidas por algoritmos rápidos. Esto quiere decir que se tiene consideración de las dependencias entre pares de variables y que solo es necesario considerar los estadísticos de orden 2. Al contrario que en los univariables, que el modelo solo tiene consideración de cada parámetro individualmente, estos pasan a considerar a todos los parámetros (la estructura). Ejemplos de este tipo de EDAs son el MIMIC (Mutual Information Maximization for Input Clustering), BMDA (Bivariate Marginal Distribution Algorithm) o Tree-EDA. Recomendados en problemas

donde las variables tengan una gran cardinalidad. Funcionan bien en una gran variedad de problemas pero son más lentos que los univariables y pueden ignorar algunas dependencias de las variables.

- **Multivariables**: conocidos alternativamente como "EDAS con dependencias multiples", este tipo de EDAs factorizan la probabilidad conjunta de distribuciones usando estadísticas de orden mayor a 2. Esto hace que la complejidad del modelo probabilístico sea mayor, al igual que el proceso de aprendizaje. La mayoría de este tipo de algoritmos usan redes Bayesianas o redes de Markov para codificar la distribución de probabilidad a cada paso. Ejemplos algoritmos multivariables son el FDA (Factorized Distribution Algorithm), EcGA (Extended compact Genetic Algorithm), BOA (Bayesian Optimization Algorithm) o EBNA (Estimation of Bayesian Network Algorithm). Según la forma en la cual el modelo es aprendido, los EDAs con dependencias múltiples se dividen a su vez en estructura+parametros, en los que se induce la estructura del modelo y sus parámetros y el aprendizaje por parámetros, en los que solo se inducen los parámetros de forma a priori. Cada uno tiene sus ventajas y desventajas, pero ambos tienen un gran consumo computacional (sobre todo los segundos, pero a cambio son mas flexibles).

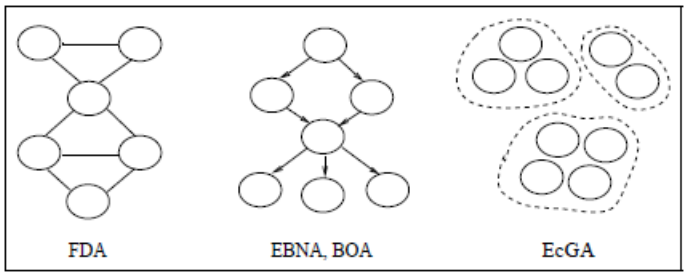


Figura 2.14: Representación gráfica de los EDAs con dependencias múltiples [21]

2.3.2. Tree-augmented Naïve-Bayes (TAN)

El algoritmo TAN (por sus siglas en ingles tree-augmented naïve-bayes) es un tipo de clasificador bayesiano de tipo semi-naïve planteado por Friedman, Geiger, y Goldszmidt en 1997 [22]. Este clasificador tiene en cuenta las dependencias entre las variables, al contrario que el método original Naïve-bayes, el cual hace la asunción "naïve"de que cada atributo es independiente del resto dada la clase, lo cual es contrario a lo que ocurre normalmente en la realidad.

El método recibe ese nombre debido a la estructura de grafo que utiliza pero al contrario que su versión original, el TAN añade nuevos arcos al grafo permitiendo que se capture la correlación entre los atributos (la interdependencia entre ellos), lo que conlleva un aumento del coste computacional. Es por ello que además el TAN impone restricciones al nivel de interacción entre variables (que de forma natura cada atributo puede estar relacionado con varios), siendo esta restricción de una sola interacción entre variables y todas las variables están conectadas con las variables de clase mediante un arco directo (se puede ver en la

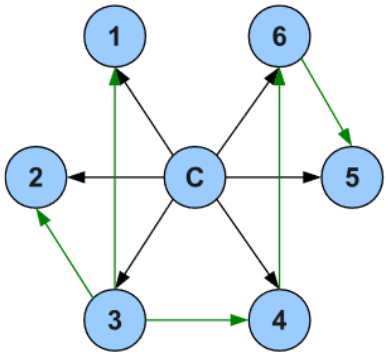


Figura 2.15: Ejemplo de un modelo TAN

figura 2.15). Matemáticamente, se denota mediante la expresión $P(X|A_1A_2...A_N)$, siendo C el nodo clase y A_i los nodos atributos. Todo esto permite que el TAN tenga mayor precisión que el modelo original. La figura 2.16 muestra un ejemplo práctico de un modelo TAN para el ámbito de la evaluación de un coche.

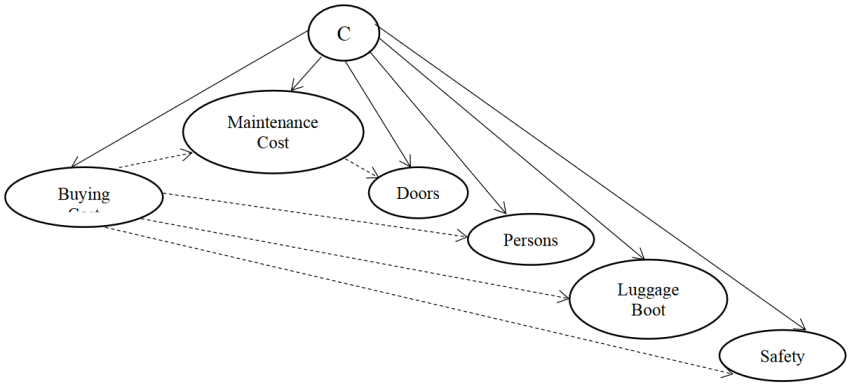


Figura 2.16: Ejemplo de un modelo TAN para la evaluación de un coche [23]

Se observa en la figura que el nodo clase está directamente conectado con todos los nodos atributos mediante un arco sólido mientras que cada nodo atributo esta conectado con otro nodo atributo mediante un arco con puntos intermitentes. Nótese que cada variable en el grafo puede tener dos nodos padres, exceptuando el nodo raíz que en el caso de la figura 2.16 es el nodo "Buying Cost" (nodo que está más a la izquierda). Para construir la estructura se debe averiguar para todo nodo, cual es que el está más relacionado con él, para poder añadir el arco (para que sea su padre). Esto hace que el modelo cambie de un grafo a un estructura de tipo árbol. En resumen, los TAN son una versión mejorada de los Naïve-Bayes y funcionan bien tanto en datasets grandes y pequeños.

Para este trabajo se utilizará el método TAN como la clase del modelo probabilístico que subyace al EDA y que se explica más adelante en el apartado 3.2.2.3.

2.3.3. Métodos de Computación de Alto Rendimiento

A lo largo del estado del arte se han mencionado distintos métodos algoritmos para poder trabajar con problemas de decisión. Aun así en la mayoría de casos, debido al gran tamaño que tienen estos problemas, es insuficiente usar métodos heurísticos ya que los tiempos que se necesita para evaluar los modelos son muy altos [24]. Es por ello que se necesitan métodos de programación que usen el paralelismo que ofrecen las CPUs de los ordenadores. Estos métodos son esenciales no solo en el ámbito de la teoría de decisión sino también en otras áreas del mundo científico.

Debido a la gran variedad de métodos de computación de alto rendimiento, como la vectorización, paralelización, uso de GPUs (Unidad de Procesamiento Gráfica) o sistemas distribuidos, en este trabajo se va a usar las herramientas de paralelización que ofrece el lenguaje R y además se va a intentar mejorar los accesos de memoria en el código desarrollado. El lenguaje R, que funciona sobre el lenguaje C, trabaja bien sobre un solo procesador, pero a veces, se necesita alcanzar mejores prestaciones o se hace un uso intensivo de la memoria.

En lo relativo al paralelismo se trata de dividir la carga computacional que normalmente se ejecutaría en un procesador en hilos ligeros (threads) o hilos pesados (en otros procesadores). Con todo esto, hay que tener cuidado a la hora de paralelizar, debido a numerosos factores:

- Dependencias en el código paralelizado: si usas por ejemplo 4 hilos, el *speedup* no tiene porque ser de 4, ya que puede haber ciertas partes del código que no se puedan paralelizar o pueden causar dependencias, haciendo que aunque haya varios hilos en ejecución estos se ejecuten secuencialmente.
- Granularidad del código paralelizado: hay que estudiar que como se paraleliza, si pequeñas líneas de código o grandes bloques de código.
- Sobrecoste de arrancar el paralelismo: hay un coste asociado al arrancar el paralelismo, siendo a veces contraproducente paralelizar el código, especialmente si se trabaja con problemas pequeños o en secciones donde no se puede mejorar al código secuencial.

Es por ello que se debe de estudiar que partes del código es posible paralelizar además de observar que zonas son cuellos de botella en el programa. Debido a esto es necesario utilizar herramientas como los profiler. Estos programas permiten medir que áreas del código son ineficientes o tienen un gran consumo de CPU y de memoria, uso de recursos, información de las llamadas a funciones realizadas, etc. En el lenguaje de R, el entorno de desarrollo de RStudio trae un profiler, que realiza estas funciones. Una vez que sabemos que partes del código causan problemas, se estudia si es posible realizar mejoras para paliar los recursos que consume el programa. Una vez realizados los cambios, de forma iterativa, se vuelve a consultar al profiler.

El lenguaje R trae varias herramientas para paralelizar el código como las funciones *lapply* o *mcapply*, que distribuye una función a varios procesadores y devuelve los resultados en forma de lista, o la librería *doParallel*, que funciona

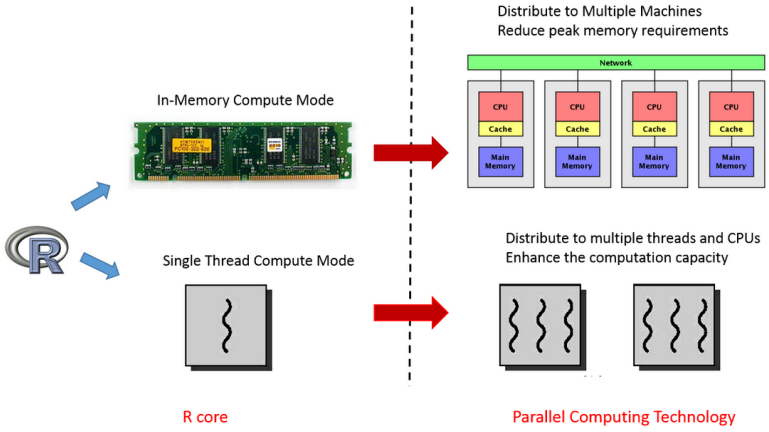


Figura 2.17: Aproximaciones de la paralelización en R

en paralelo con el *foreach*. Esta librería permite paralelizar bucles *for* mediante la estructura *foreach* y poder especificar cuando procesadores usar y de que forma se devuelven los resultados.

Capítulo 3

Desarrollo

Este capítulo presenta el problema que se trata en el trabajo y la solución propuesta, describiendo el proceso paso a paso y finalizando con las pruebas para para validar los resultados. Primero, se plantea el problema y la solución que se ha seguido. Luego se muestra el código que se ha desarrollado para el trabajo, junto con los métodos de optimización empleados. Por último, se describen las pruebas que se han hecho para poder validar el código.

3.1. Planteamiento, estudio del problema y su solución

Como ya se ha ido explicando a lo largo de la memoria, la complejidad de los problemas que han ido abordando los DSS han ido creciendo a lo largo de los años, pero también así los recursos computacionales de los que se dispone. Aun así, muchas veces es complicado satisfacer la demanda computacional de estos problemas.

Pongamos por ejemplo el caso del modelo IctNeo (mencionado en el apartado 2.1.3 en el que ha trabajado el Grupo de Análisis de Decisiones y Estadística del departamento de Inteligencia Artificial de la Universidad Politécnica de Madrid. Este modelo, que se puede ver en la figura 3.1 (que a su vez es una versión reducida del estudiado en su momento), cuenta con dos nodos decisores, un nodo utilidad y 27 nodos probabilísticos. Su tabla de decisiones asociada tiene 1990655 filas y se necesitan de varios minutos para poder ejecutar el algorítmo. Si evaluamos exclusivamente el primer nodo decisor, la complejidad del problema se reduce a una tabla de 1728 filas, de las cuales 432 son decisiones óptimas.

En casos de modelos más grandes (y normalmente son aquellos que tratan problemas reales), se hace necesario el uso de ordenadores con grandes prestaciones o incluso superordenadores. También hay que pensar en la memoria que ocuparían estas tablas. Si un tipo entero (integer) en R son 4 bytes y tenemos, usando el ejemplo del modelo previo, 29 columnas y 1990655 filas, hacen cantidades de memoria elevadas, lo cual nos obliga a pensar en diversas soluciones para poder evaluar el modelo.

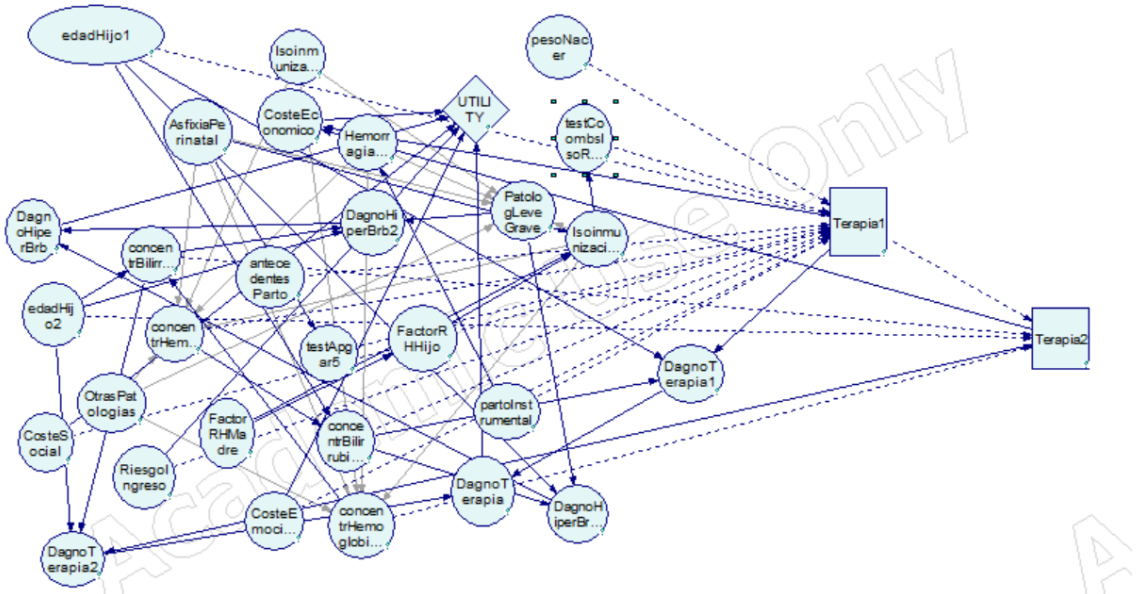


Figura 3.1: Diagrama de influencia IctNeo [11]

Una posible solución puede ser restringir el dominio del problema, por ejemplo, considerar aquellos casos del mismo que sean más frecuentes descartando los menos frecuentes, consiguiendo así que el problema sea menos complejo. Sin embargo, esto puede llevar a reducir el conocimiento que tiene el modelo sobre el problema y pudiendo reducir su solidez a la hora de dar soluciones.

Otra posible solución, y es la que se ha elegido para este trabajo, es el uso de las listas KMB2L, cuyo objetivo es la de reducir el tamaño que ocupa el modelo sin alterar la estructura del mismo. Gracias a las KMB2L, pasamos de tener una **tabla de decisiones óptimas** con N atributos a una tabla o matriz de 2 columnas que cuenta con el desplazamiento y la propia decisión óptima. Con estas listas podemos reducir sustancialmente el tamaño que ocupan estos modelos, sin alterar el conocimiento que se tienen de los mismos.

El otro problema presente en este trabajo, asociado a la búsqueda de la lista KMB2L óptima, es el problema combinatorio de permutación de columnas. Las filas se permutan de forma solidaria con las columnas, que definen el índice que ordena la tabla. Este problema es NP-Completo y escala de forma exponencial según el número de columnas y el tamaño de la tabla, que depende del cardinal del dominio de los atributos-columnas. El uso de métodos tradicionales, como la búsqueda exhaustiva puede funcionar cuando se presentan problemas pequeños, pero a medida que se presentan modelos con mayor envergadura se pueden exceder las capacidades computacionales (tanto de memoria como de capacidad de computo) de la máquina.

Es por ello que se hace necesario el uso de métodos heurísticos para poder lidiar problemas de gran tamaño y en este trabajo se han usado los llamados Algoritmos de Estimación de distribución (EDAs), que han demostrado ser muy

útiles y sencillos en aplicaciones del mundo real.

Finalmente, las personas que toman decisiones demandan que los DSS sean accesibles (cosa que las tablas de decisión permiten fácilmente) y que la decisión que se proporcione venga asociada a una explicación que diga los motivos por los que esa decisión es óptima, lo cual también sirve para validar el modelo. Esto, como se explica en este capítulo, es resuelto también por las listas KBM2L.

3.1.1. Listas KBM2L y su construcción

Las listas KBM2L (en ingles, Knowledge Base Matrix To List [2]) son un tipo de representación de las bases de conocimiento en forma de lista, cuyo objetivo es el de reducir el tamaño que ocupan estas bases de conocimiento sin perder el saber que se tiene del modelo.

Partiendo de la base de conocimiento y el conjunto de atributos, el cual se denomina **esquema**, definimos un orden de los mismos. A ese orden de atributos del esquema se le denomina **base**. Las variables del modelo son categóricas y para poder ser representadas toman valores discretos, con lo cual estamos asumiendo un orden en el dominio. Por ejemplo, se puede definir la variable "Nota Alumno", que toma valores de 0 a 10 (siempre enteros). El orden se infiere en que una nota 0 es menor que 1 y así sucesivamente. Este orden puede ser natural (como el presentado) o convencional.

Con ello, podemos definir el concepto de **índice**, que no es mas que un vector que contiene valores que representan coordenadas con respecto a la **base**. Por ejemplo, el índice i accede a la fila j de la tabla (similar a los índices de un dataframe). Gracias a imponer un orden en el esquema (que deberá ser siempre ascendente, en función de los valores de los atributos) y con la ayuda de un vector de pesos, que indica la importancia de los atributos, se puede definir el problema como matrices multidimensionales (MMs). La matriz tiene dos vistas: en n dimensiones (arrays) y en 1 dimensión (vectores). El índice y el peso de cada dimensión permite calcular el ordinal de cada celda en 1 dimensión. Explicamos en detalle la forma de serializar las matrices para construir las listas KBM2L.

Dados los atributos en cierto orden, $\vec{w} = (w_0, w_1, ..., w_n)$ y el vector de coordenadas $\vec{c} = (c_0, c_1, ..., c_n)$, se denota D_i como el producto escalar del dominio, y $w_i = \prod_{j=i+1}^{n} D_j$. Con todo ello se define el *offset (q)* como el producto de las coordenadas con los pesos, que es el desplazamiento u **offset** del elemento con respecto al primer elemento de la tabla de una base respectiva. Este concepto es similar al manejo de estructuras de datos en las memorias de los ordenadores, ya que generalmente es una estructura de acceso lineal, que mediante punteros a direcciones físicas, permite implementar estructuras no lineales, arrays, árboles, grafos...

Al primer elemento del vector de pesos se le asigna el valor 1 (menor peso posible, ya que su dominio es 1), ya que va asociado al atributo de menor importancia de la *base* (el que está más a la derecha de la misma) y sucesivos pesos se calculan como las combinaciones de los dominios de los atributos previos. Este cálculo se puede deshacer, ya que si tenemos el offset podemos conocer el índice

asociado. El cambio de base hace que se cambien los pesos y los índices y, por consiguiente, el offset de los elementos de la tabla, ya que siempre está ordenada de forma ascendente. Y de este modo permutan las celdas de la MM vista como vector, cambiando la adyacencia entre celdas.

Una vez que se tienen el *offset* de cada elemento de la base, solo hace falta conocer el último elemento de cada bloque de una misma decisión para generar la lista KBM2L. Se pueden ver las KBM2L como listas sparse de decisiones o como un número reducido de etiquetas de las alternativas de la decisión, y se distribuyen en un vector de tamaño combinatorio. Siguiendo el ejemplo de la figura 3.1, en concreto, en la del primer nodo decisor (Terapia 1), su lista KBM2L contiene 32 *items* y se puede ver su representación matricial en R en la imagen 3.2.

Esta lista denota lo siguiente, que entre el offset 0 y el 11 hay 12 1's, del 12 al 23 hay 11 0's y así sucesivamente hasta el offset N, que denotará el tamaño de la lista. Esto hace que nos ahorremos mucho espacio de almacenamiento y no se está alterando el conocimiento del modelo.

	reduced_arr	offset_arr
1	1	11
2	0	23
3	2	47
4	1	59
5	3	71
6	1	95
7	0	107
8	1	119
9	0	131
10	2	143

Figura 3.2: 10 primeros items KBM2L IctNeo primera decisión

Esta lista que se obtiene se puede optimizar, y es que, dependiendo del orden de los atributos de la *base*, los *items* o elementos de la lista pueden ser menores que los de otra *base*. El problema de optimización de permutaciones asociado es, en un espacio de tamaño n, $n!$, y probar si una solución es mejor que otras es k^n, donde k es el cardinal medio de los atributos. Luego la búsqueda de la lista óptima es un problema NP-completo, ya que se tienen que probar todas las posibles combinaciones de columnas para que se obtenga una lista con unos *items* que no estén fragmentados. Además, a esto se suma que los problemas asociados a la creación de nuevos espacios de memoria para las*bases* generadas.

En la imagen 3.3 se puede ver que el conjunto de posibles soluciones (del 0 al 4, representación discreta, el tamaño del bloque indica la decisión, siendo el 0 una línea) está muy fragmentado.

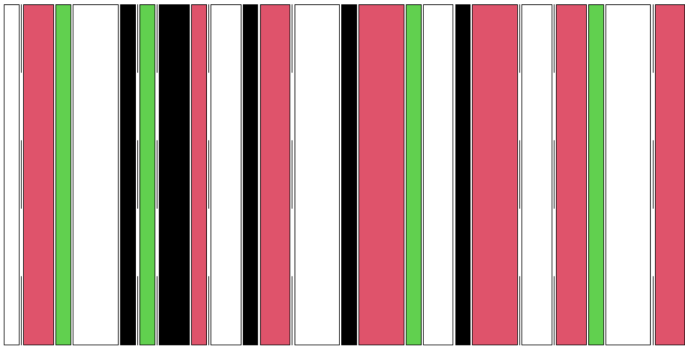


Figura 3.3: Fragmentación primera decisión IctNeno (espectro lista)

3.1.1.1. Ejemplo proceso construcción lista KBM2L

En esta sección se describe el proceso de construcción de una KBM2L y se muestran las notaciones matemáticas en las que se sustenta. Para ello, se presenta la siguiente tabla óptima, que servirá de ejemplo en para la creación de la KBM2L.

La tabla 3.1 trata un problema de decisión simple, que se enmarca en la decisión de un trabajador de cómo ir al trabajo según unos antecedentes. En este caso, los antecedentes son el tráfico en la carretera (bajo o alto), situación transporte público (sin averías o con averías) y nivel de gasolina del vehículo del trabajador (bajo o alto). La decisión en este caso es 0 si va en coche, 1 si va en transporte público y 2 si teletrabaja. Notese que, para el orden de los valores cada atributos está impuesto.

Expresando el ejemplo en notación matemática, el esquema sería Tráfico, Avería, Transporte, Nivel Gasolina y Decisión. La base, en el caso de la tabla 3.1, sería

$$B_0 = [0, 1, 2]$$

siendo 0 tráfico, 1 avería y 2 gasolina.

Tráfico	Avería Transporte	Nivel Gasolina	Decisión
0	0	0	0
0	0	1	0
0	1	0	2
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	2
1	1	1	2

Cuadro 3.1: Ejemplo tabla de decisiones óptimas

La tabla óptima modeliza las preferencias del decisor, en este caso el trabajador, al igual que en un problema real. Los atributos se han codificado como binarios para que el problema sea lo mas sencillo posible.

Esta misma tabla puede ser representada con una lista de tuplas que contengan la posición (offset) y la decisión. Por ejemplo, en el caso de la tabla del trabajador la lista sería la siguiente:

$$< (0,0) < (1,0) < (2,2) < (3,0) < (4,1) < (5,1) < (6,2) < (7,2)$$

que se puede abstraer a cualquier tabla mediante la siguiente expresión,

$$< (0,x) < (1,y) <, ..., < (N-1,z)$$

siendo x,y,z las políticas o decisión óptima mientras que N denota el tamaño de la tabla. El símbolo «"denota el orden de los elementos de la lista. Esta notación se la denomina "notación offset".

La lista KBM2L asociada a la tabla 3.1 quedaría de la siguiente forma:

$$< 1,0| < 2,2| < 3,0| < 5,1| < 7,2|$$

Cada elemento de la lista se le denomina **item** y en este caso la lista tiene 5 items (frente a los 8 elementos de la notación offset). Existe una notación alternativa denominada notación de índices en el cual en vez de usar el offset se utiliza las instancias de los atributos:

$$< (0,0,1),0| < (0,1,0),2| < (0,1,0),0| < (1,0,1),1| < (1,1,1),2|$$

La notación (offset, decisión) refleja dos ideas, que el offset es siempre creciente y que permite sintetizar un conjunto de celdas contiguas que tienen la misma decisión en una misma celda, reduciendo el tamaño de la lista sin alterar el conocimiento que se tiene sobre el modelo. La lista KBM2L es una versión comprimida del la MM, pero sin perdidas como el *algoritmo zip*.

Inicialmente, la lista está vacía. Esto se denota con el item $< w_0 D_0 - 1, -1|$, y el -1 denota la ausencia de conocimiento sobre el modelo. Cuando se tenga más conocimiento del modelo y nuevos items, estos se insertan en la lista mediante los offsets. Por ejemplo, si tenemos el item $< q, d|$, la inclusión se realiza comparando los offsets de los items previos de tal forma que

$$q_{i+2} < x < q_{i+3}, x - 1 = q_{i+2}, d \neq d_{i+2}, d \neq d_{i+3}$$

Esta expresión dice que el offset del nuevo item se debe de introducir entre dos items tales que la decisión sea distinta y que el offset sea mayor que el que está a su izquierda y menor que a su derecha.

Ahora que se tiene la lista inicial, se busca un cambio de base tal que la nueva base tenga menos items que la lista inicial. Para ello, se permutan los atributos de la base. Siguiendo el ejemplo de la tabla 3.1, permutamos las columnas de tráfico y nivel de gasolina, cambiando de la base inicial B_0 a la siguiente base:

$$B_1 = [2,1,0]$$

y la nueva tabla, que recordemos, debe de estar ordenada de manera ascendente en su dominio.

Nivel Gasolina	Avería Transporte	Tráfico	Decisión
0	0	0	0
0	0	1	1
0	1	0	2
0	1	1	2
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	2

Cuadro 3.2: Nueva Base Problema

La nueva base, que queda reflejada en la tabla 3.2, tiene asociada la siguiente lista KBM2L:

$$< 0,0| < 1,1| < 3,2| < 4,0| < 5,1| < 6,0| < 7,1|$$

lo cual es un empeoramiento con respecto a la base inicial, ya que esta cuenta con 7 items frente a los 5 de B_0. El número de posibles bases viene en función del número de atributos del problema, en este caso son un total de 6 bases (3!).

La idea general es colocar aquellos atributos con menos relevancia a posiciones con menos peso para que el numero de items de la lista generada sea menor. Esto se consigue gracias a que se simplifican los bloques de una misma decisión en un solo item. Sin embargo, la búsqueda de la mejor base es un problema muy complejo y costoso a nivel computacional. En el caso del ejemplo son pocas las bases, pero en problemas más grandes, el número de bases escala de forma factorial.

Existe una opción en problemas donde el conjunto de atributos es muy grande: se fijan pesos constantes a algunos atributos y permutar el resto de atributos en búsqueda de una mejor base. Esto es parecido a trabajar con un subproblema del problema original, siendo la probabilidad de encontrar la base óptima igual a $\frac{1}{n!}$ (n el conjunto de atributos del problema). En el caso de que haya x atributos relevantes, esa probabilidad es de $\frac{x!}{n!}, x < n$.

Por último, todas las bases del modelo pueden ser representadas en un grafo en el que cada nodo es una base y el arco de un nodo a otro (que puede ser bidireccional) representa la permutación necesaria para llegar a esa base. Esto vuelve a resaltar la naturaleza del propio problema y la necesidad de utilizar métodos heurísticos para encontrar el nodo en el grafo cuya KBM2L sea la de menor tamaño.

Siguiendo el ejemplo planteado en la tabla 3.1, que tiene 3 atributos, se puede representar el conjunto de bases del problema en el siguiente grafo personificado en la figura 3.4. En ella, la base inicial se denota con B_0 (base azul) y las bases en las que solo se necesita una permutación de atributos para llegar a ella se representan en color verde. Las que se necesitan 2 permutaciones se denotan con un color naranja.

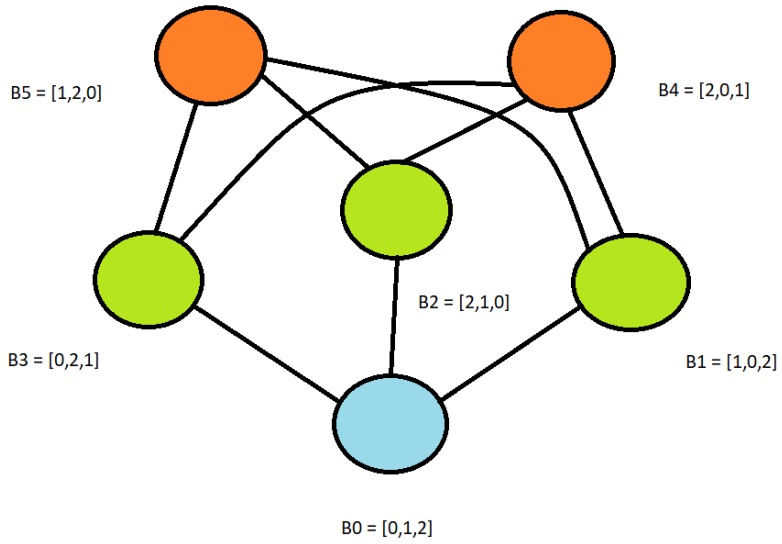


Figura 3.4: Grafo de las bases del problema del transporte

3.1.1.2. Explicabilidad de las KBM2L

Como se ha reiterado, una de las consecuencias del uso de las listas KBM2L son la reducción de tamaño del modelo, pero también se consiguen beneficios en la explicabilidad del DSS. Esto se debe a que cada *item* agrupa a casos adyacentes con la misma solución, y analizando las reglas que llevan a la solución, se puede ver que atributos son iguales y cuales son diferentes, explicando la política o decisión por si misma.

De la misma forma que el método *permutation importance* descrito en el punto 2.2.2.2, al reordenar las columnas y al reducir los *items* de la lista, estaríamos dando una capa de explicabilidad al modelo, ya que, al agrupar los atributos estaríamos viendo que cambios de los mismos hacen que se cambie de decisión. Mientras que en el método permutation importance vemos los cambios al permutar los valores de un atributo, aquí estamos permutando varios atributos para reducir el tamaño de la lista e, indirectamente, viendo que atributos afectan más a la decisión.

Y aparte poder ver que atributos hacen que, al modificarse, se cambie de decisión, se puede observar también aquellos atributos que no toman parte en la decisión. Por un lado, los atributos con valores fijos dentro de un mismo *item* nos estarían indicando o explicando la influencia que tienen sobre la decisión mientras que, los atributos que varían dentro de un *item* nos muestran la irrelevancia que tienen en la política a tomar. Esto último se debe a que si se mantienen unos atributos con valores fijos y otros atributos si varían, estando dentro de un mismo *item* y por ende decisión, nos estarían mostrando que atributos impactan en la decisión y cuales no.

Toda esta idea se complementa con el uso de los algoritmos de estimación de

distribuciones, mostrados en el apartado 2.3.1.1 (EDAS), ya que la función *fitness* busca aquellas bases que tengan menores *items*, lo cual mejora la lectura de las tablas y, por consiguiente, la identificación de patrones en ellas.

Por último, podría ser útil aplicar técnicas minería de datos como clustering (clustering jerarquico, basado en distancias, etc) y reducción de dimensionalidad como PCA (Principal Components Analysis) para hacer un análisis de cada decisión y ahondar aun más en la explicación de los atributos y sus políticas.

3.1.2. Método UMDA

El algoritmo UMDA (en ingles Univariate Marginal Distribution Algorithm) fue introducido por Mühlenbei y Paaß en 1996, es un tipo de EDAs que no tiene dependencias cuya generación de la probabilidad de distribución conjunta es la más simple de todas. Esta distribución se ha factorizado como el producto de distribuciones univariantes independientes,

$$p_l(x) = p(x|D_{l-1}^{Se}) = \prod_{i=1}^{n} p_l(x_i)$$

siendo D_{l-1}^{Se} la población de la generación l y Se el número de elementos de la población. El pseudocódigo del algoritmo se puede ver en la figura 3.5.

UMDA
$D_0 \leftarrow$ Generar M individuos (la población inicial) al azar

Repetir para $l = 1, 2, \ldots$ hasta que se verifique el criterio de parada

$D_{l-1}^{Se} \leftarrow$ Seleccionar $N \leq M$ individuos de D_{l-1} de acorde con un método de selección

$p_l(\boldsymbol{x}) = p(\boldsymbol{x}|D_{l-1}^{Se}) = \prod_{i=1}^{n} p_l(x_i) =$

$\prod_{i=1}^{n} \frac{\sum_{j=1}^{N} \delta_j(X_i = x_i | D_{l-1}^{Se})}{N} \leftarrow$
Estimar la distribución de probabilidad conjunta

$D_l \leftarrow$ Muestrear M individuos (la nueva población) a partir de $p_l(\boldsymbol{x})$

Figura 3.5: Pseudocódigo del UMDA [21]

La distribución marginal juega un papel muy importante en este método ya

que permite caracterizar la independencia entre dos variables aleatorias. Se recuerda que dos variables aleatorias son independientes si y solo si la función de distribución conjunta es igual al producto de sus funciones de distribución marginales.

En el caso del algoritmo, cada distribución de probabilidad univariante se estima gracias a las frecuencias marginales:

$$p_l(x_i) = \frac{\sum_{j=1}^{N} \delta_j(X_i = x_i | D_{l-1}^{Se})}{N}$$

Donde la expresión del numerador toma el valor 1 si en el j-ésimo caso de $D_{l-1}^{Se}, X_i = x_i$ y 0 en caso contrario. N denota el número de elementos de la subpoblación escogida.

3.2. Codificación de la solución

En este apartado se muestra el proceso de construcción de la solución al problema planteado, desde el diseño de alto nivel a la propia codificación de los métodos. El proceso se divide en varias partes, siendo la primera la relacionada a la construcción de la infraestructura previa necesaria para el manejo de las KBM2L, diagramas de influencia y tablas de decisiones óptimas. La segunda fase de este apartado radica en el diseño y programación de los métodos de optimización, que abarca los distintos EDAs usados en el trabajo y las técnicas de computación de alto rendimiento.

3.2.1. Codificación de la infraestructura de las KBM2L

En esta sección se explican las distintas funciones que se han usado para montar la infraestructura necesaria para el manejo del problema, esto es, el paso de diagramas de influencias a tablas de decisiones óptimas, cambio de base y construcción de la lista KBM2L.

3.2.1.1. Diagramas de influencia a tablas de decisión

Inicialmente, se trabaja con diagramas de influencia, que sera nuestro input para las funciones que se van a implementar en este apartado. En este primer paso, que consiste en pasar de diagramas de influencia a tablas de decisión, se utiliza la librería IdR (Influenca Diagrams on R), desarrollada por Juan Antonio Fernandez del Pozo y Concha Bielza para el lenguaje R [25].

Esta librería nos permite representar problemas de decisión en R de forma gráfica usando diagramas de influencia y redes bayesianas de forma sencilla además de poder evaluar estos modelos y hacer análisis de sensibilidad y la exactitud de los mismos. Además permite exportar los modelos a otros software relacionados con la materia como GeNIe (desarrollado por BayesFusion [10], software con GUI que tiene funcionalidades similares que la librería IdR), además de que

la librería está especialmente optimizada, debido al gran coste computacional de las operaciones que realiza.

Si nos centramos en IdR [25], la base principal para construir los modelos de los problemas de decisión es el nodo o "node", que nos permite especificar elementos del problema de decisión tales como el tipo de nodo, antecedentes o consecuentes. Un ejemplo de una implementación de un problema de decisión se puede ver en el siguiente código escrito en lenguaje R [4]:

Listing 3.1: Ejemplo de codificación modelo mediante librería IdR

```
## Influence Diagram

helicopter = list(

Resultados = node( Type = "CHANCE", Name = "Resultados", Values = c("Apto","No.Apto","
    Nada"), Preds = c("Prueba","Estado"),
Pots = matrix( data = c(
0.90, 0.10, 0.0,
0.65, 0.35, 0.0,
0.15, 0.85, 0.0,
0.0, 0.0, 1.0,
0.0, 0.0, 1.0,
0.0, 0.0, 1.0),
  nrow = 6, ncol = 3, byrow = TRUE, dimnames = NULL)),

Estado = node(Type="CHANCE", Name="Estado", Values = c("Muy.maniobrable","Media.
    maniobrabilidad","Poco.maniobrable"), Preds = c(),
Pots=matrix( data = c(
0.30,0.40,0.30),
  nrow=1,ncol=3,byrow=TRUE,dimnames=NULL)),

Prueba = node(Type="DECISION", Name="Prueba", Values = c("Probar","No.Probar"), Preds =
    c(),
Pots = matrix( data = c(1.0), dimnames = list("phase","Prueba"))),

Compra = node(Type="DECISION", Name="Compra", Values = c("Comprar","No.Comprar"), Preds
    = c("Prueba","Resultados"),
Pots = matrix( data = c(2.0),dimnames=list("phase","Compra"))),

UTILITY = node(Type="UTILITY", Name = "UTILITY", Values=c(0.0,1.0), Preds = c("Compra","
    Estado","Prueba"),
Pots = matrix( data = c(
0.85,0.43,0.0,0.25,0.25,0.25,0.86,0.44,0.01,0.26,0.26,0.26),
  nrow=12,ncol=1,byrow=TRUE,dimnames=list( NULL, c("UTILITY"))))
)
cat("Influence Diagram -- helicopter: ", names(helicopter),"\n")
```

Este problema tiene dos nodos de decisión, el primero indica la decisión de probar o no el helicóptero y el segundo el de comprarlo o no. Se puede saber gracias al parámetro de "node" llamado "type" que puede tomar los siguiente valores ["chance,decision,utility"]. También con el parámetro "preds" podemos ver los nodos antecedentes. Con "values", los valores que puede tomar el nodo. Por último, se define una matriz de valores de probabilidad en función del valor que toma el nodo y los nodos previos.

Una vez que tenemos el modelo construido como una lista de nodos, creamos el diagrama de influencia con la función del paquete IdR llamada `influence.diagram()`, que recibe como argumento la lista de nodos del modelo. Luego usaremos

la función `arcrevalg.eval()`, cuya misión es la de evaluar el modelo, generando la tabla de decisiones con las utilidades. Esta función implementa el algoritmo de inversión de arcos de Shacter 1986. Por último, el paquete IdR puede transformar el diagrama de influencia generado a un formato que puede ser leído por otros software como GeNIe mediante la función `dump.netG()`. En la figura 3.6, se puede ver el resultado de transformar el modelo del helicóptero a GeNIe. Ahora que tenemos la tabla de decisiones con las utilidades, el siguiente paso es calcular la tabla de decisiones óptimas.

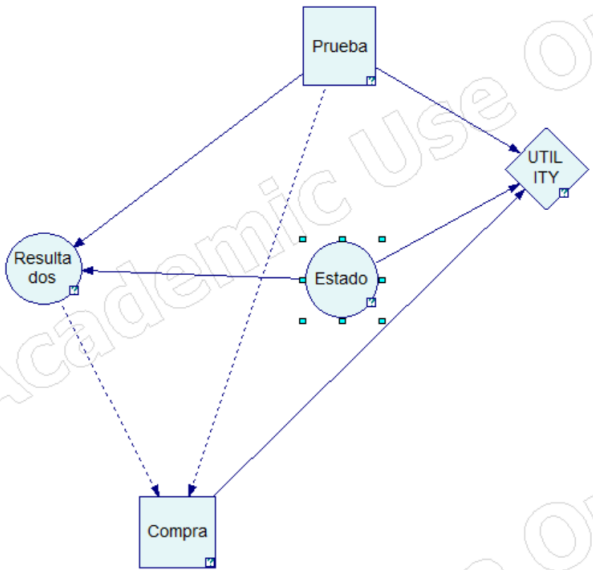


Figura 3.6: Modelo helicóptero en el interfaz de GeNIe

Por último, resaltar que desde GeNIe se puede obtener también la tabla de decisiones y que se puede usar el resultado como input de las funciones que se van a presentar. La desventaja que tiene GeNIe con respecto a IdR es que este primero escala mal con modelos de tamaño mediano y grandes (al menos en su versión gratuita que se dispone), incluso siendo imposible evaluar el modelo.

3.2.1.2. Tablas de decisión a tablas óptimas

Una vez que hemos evaluado el diagrama de influencia y tenemos la tabla de decisiones óptimas con sus utilidades, transformaremos esa tabla a una que solo disponga de las decisiones óptimas (el paquete *IdR*, además de la decisión óptima también muestra el resto de utilidades).

Como se ha explicado en el apartado 2.1.2.3, las tablas de decisiones contienen las reglas a la hora de tomar la decisión. Tras usar las funcionalidades del paquete IdR, se tiene además de la tabla de decisiones, sus utilidades, por lo que si queremos tener las decisiones óptimas solo tenemos que escoger aquella decisión con utilidad máxima.

Siguiendo el ejemplo de la figura 3.7,que representa un problema de decisión que

	CosteSocial	CosteEmocional	RiesgoIngreso	testApgar5	partoInstrumental	testCoombsIsoRHHijo	Terapia1	Utility
1	0	0	0	0	0	0	0	0.6561222
2	0	0	0	0	0	0	1	0.7273500
3	0	0	0	0	0	0	2	0.7267885
4	0	0	0	0	0	0	3	0.7061542
5	0	0	0	0	0	1	0	0.6561222
6	0	0	0	0	0	1	1	0.7273500
7	0	0	0	0	0	1	2	0.7267885
8	0	0	0	0	0	1	3	0.7061542

Figura 3.7: Tabla de decisiones con utilidades (primeras filas)

tiene 4 posibles alternativas, seleccionamos de cada bloque de las 4 decisiones aquella que tenga mayor utilidad. Tenemos tantos bloques como $\frac{N}{D}$ siendo N el tamaño de la tabla de decisión y D el número de posibles decisiones. Una vez que se haya elegido la mejor alternativa de cada bloque, tenemos la tabla óptima. El código en lenguaje R desarrollado para esta sección se encuentra disponible en el anexo .2.3 y a continuación se muestra un pseudocódigo del método (pseudocódigo1).

Algorithm 1 Decision Table to Optimal Decision Table

Require: dfx_{nm}
  $d \leftarrow nDecisions$ ▷ Número posibles decisiones
  $nBlocks \leftarrow \frac{dfx_{n:}}{d}$ ▷ Obtienes el número de bloques de la tabla de decisión
  $i \leftarrow 0$
  $j \leftarrow 0$
  while $i < n$ **do**
    $block_i \leftarrow dfx_{i:i+d,:}$
    $new_dfx_{j:} \leftarrow block_{\text{máx}\, i,m}$ ▷ Obtienes la máxima utilidad de cada bloque
    $i \leftarrow i + d$
    $j \leftarrow j + 1$
  end while

3.2.1.3. Manejo de Bases

Esta sección trata de mostrar las distintas funciones necesarias para el manejo de bases. Ahora que se tiene la tabla de decisiones óptimas, se considera la base inicial como el orden de atributos inicial que presenta la tabla de decisiones óptimas. Las funciones que se presentan en este apartado están disponibles en el anexo .2.

Para poder manejar las bases, necesitamos por un lado una función que permita cambiar el orden de las columnas y por ende, de base. Es vital esta función ya que nos permitirá encontrar la base cuya KB2ML asociada tenga el número menor de *items*. Los argumentos de la función son la columna que se quiere permutar y hacia donde se quiere permutar (origen, destino). Se presenta el pseudocódigo de esta función (pseudocódigo 2):

Por otro lado, se necesita la función que calcule la KBM2L asociada a la base.

Algorithm 2 Swap Columns Order

Require: $i \geq 0, j \geq 0, i < m, j < m, i \neq j, dfx_{nm}$
$dfx_{:i} \leftarrow dfx_{:j}$
$dfx_{:j} \leftarrow dfx_{i:}$
Order dfx by first column

Este método sigue la explicación del apartado 3.1.1.1. Una vez que tengamos el número de *items* asociada a la KB2ML de la base, podemos ir descartando o no las bases que vayamos generando. El argumento de esta función es el dataframe (del lenguaje R) asociado a la base o simplemente la columna de las decisiones y se devuelve la KB2ML correspondiente. Se igual manera, se presenta el pseudocódigo de la función (pseudocódigo 3):

Algorithm 3 Función reduce pseudocódigo

Require: $dfx_{nm}, n \geq 2$
$decision \leftarrow dfx_{:m}$ ▷ La columna de la decisión final siempre es la última
$i \leftarrow 0$
$j \leftarrow 0$
while $i \neq n$ **do**
  if $decision_i \neq decision_{i-1}$ **then**
    $reduceVector_j \leftarrow decision_{i-1}$
    $reduceOffset_j \leftarrow i - 1$
    $j \leftarrow j + 1$
  end if
  $i \leftarrow i + 1$
end while
$reduceVector_j \leftarrow decision_n$ ▷ Se añade el último elemento
$reduceOffset_j \leftarrow n - 1$
$retMat \leftarrow reduceVector \times reduceOffset$ ▷ Se devuelve una matriz de dos columnas

3.2.2. Codificación de los métodos de optimización

En esta sección del desarrollo del trabajo se explican los distintos algoritmos que se han implementado para resolver el problema de encontrar la KB2ML óptima.

3.2.2.1. Codificación del método exhaustivo

Basándonos en la teoría explicada en la sección 3.1.1.1, en el que se describe que las bases del problema se pueden expresar como una estructura de tipo grafo, y por ende, se pueden emplear técnicas de recorridos de grafos para encontrar la base óptima. En esta sección se expone una posible implementación que usa este tipo de aproximación para la solución del problema.

Se ha programado un método exhaustivo el cual se calculan todas las listas KBM2L de las bases vecinas a la base actual y se escoge para la siguiente iteración aquella que tenga un tamaño de lista menor (en este caso la primera de

ellas, en caso de que haya empate). Aunque este algoritmo se haya enmarcado dentro de los métodos de optimización, en realidad no pertenece a los mismos ya que aunque como se va a explicar a continuación use de heurísticas para ir podando el grafo, este método pertenece al conjunto de algoritmos de recorrido de grafos como el *algoritmo de Dijkstra*[26].

Se plantea como una función recursiva, en la que se usa una lista de bases ya visitadas. Esta lista consigue que se evite calcular KB2ML de las bases por las que se hayan visitado, algo muy común en esta aproximación. Una vez que se haya obtenido la mejor base de las vecinas se repite el proceso de nuevo hasta que no se encuentre una base mejor, siendo ese caso la condición de parada del algoritmo recursivo. El pseudocódigo 4 muestra el método exhaustivo.

Algorithm 4 Algoritmo exhaustivo

Require: $baseInicial, dfx_{nm}$
  $iniciarListaBasesCalculadas$
  $k \leftarrow 0$
  $kbm2lInicial \leftarrow reduce(dfx_m)$
  $ListaBasesCalculadas_k \leftarrow baseIncial$
  $k \leftarrow k + 1$
  $i \leftarrow 0$
  $j \leftarrow 0$
  while $i < m$ **do**
    while $j < m$ **do**
      if $i = j$ **then**
        $nextIter$
      end if
      $newBase \leftarrow swap(i, j, baseInicial)$ ▷ Llamada a código 2
      if $newBaseisinListaBasesCalculadas$ **then**
        $nextIter$
      end if
      $ListaBasesCalculadas_k \leftarrow newBase$
      $k \leftarrow k + 1$
      $newKbm2l \leftarrow reduce(newBase_m)$ ▷ Llamada a código 3
      if $length(newKbm2l) < length(kbm2lInicial)$ **then**
        $posibleOptimal \leftarrow newKbm2l$ ▷ Nueva posible base óptima
      end if
    end while
  end while
  if $posibleOptimal \neq Null$ **then**
    $return(algoritmoExhaustivo(posibleOptimal))$ ▷ Se vuelve a llamar al método
  else
    $return(baseInicial)$ ▷ Condición de parada
  end if

3.2.2.2. Codificación del algoritmo UMDA

Una vez que conocemos la teoría de los EDAs y la explicación del método UMDA (apartado 3.1.2), se procede con la implementación del propio algoritmo. Este se divide en 4 grandes métodos o pasos, que describen a continuación cada uno de ellos de forma extendida y al final se presenta el pseudocódigo del mismo:

1. Creación de la matriz de frecuencias, que dependerá del número de atributos del problema, de tipo matriz cuadrada y que denota como de probable sea que el atributo i esté en la posición j de la base. Inicialmente esta probabilidad es equiprobable para cada atributo y posición de la base.

2. Una vez dentro del bucle del algoritmo se debe, para cada iteración, generar la población, que utilizará la matriz de frecuencias que se ha creado y actualizado según cada iteración. El número de elementos o bases generadas dependerá de un hiper parámetro que define el tamaño del resultado.

3. Ahora que se tiene la población, se debe evaluar los elementos que la conforman con la función *fitness*, que consiste en el cálculo de la lista KBM2L y conocer su número de *items*. Una vez que tenemos el tamaño de cada lista, se ordena a la población de menor a mayor tamaño de la lista y se trunca la población, devolviendo un subconjunto de la población inicial que tiene menos *items* en la lista KBM2L asociada.

4. Se vuelve a actualizar la matriz de frecuencias, esta vez con la nueva población evaluada en la función *fitness*. Ahora la nueva matriz estará ajustada a las posiciones que toman los elementos de cada base de la población y por consiguiente, las nuevas generaciones de elementos obtendrán mejores resultados en la función objetivo. Se vuelve de nuevo al punto 2 hasta que se acabe el número de iteraciones, que es un hiper parámetro del problema.

Algorithm 5 Método EDA de tipo UMDA

Require: $nIter, nPop, nTrunc, nAttr$ ▷ Hiper parámetros del algoritmo
  $M \leftarrow generateMatrixDistribution(nAttr)$
  $i \leftarrow 0$
  while $i < nIter$ **do**
    $newPopulation \leftarrow generateNewPopulation(M, nPop)$
    $fitnessPopulation \leftarrow fitnessFunction(newPopulation, nTrunc)$
    $M \leftarrow updateMatrixDistribution(M, fitnessPopulation)$
    $i \leftarrow i + 1$
  end while

La función objetivo, a su vez se descompone en varios métodos, ya que por un lado se debe obtener la base B_i de la población generada y calcular su lista KBM2L, por lo que se utilizan los métodos construidos en el apartado de la infraestructura. Estos permiten cambiar de base y construir la lista y finalmente se devuelve la población ordenada con mejor *fitness* (esto es el número mínimo de *items* asociado a la base que configura la lista KBM2L), que es un subconjunto de la población inicial.

3.2.2.3. Codificación del algoritmo TAN

En el apartado 2.3.2 se ha explicado que el Tree-Augmented Naïve, TAN, es un tipo de clasificador bayesiano, luego para poder usarlo en nuestra problema se debe complementar con un EDA. Partiendo de la base del UMDA ya desarrollado, en vez de utilizar una distribución que genere la nueva población, se utilizará el TAN para generar por un lado el árbol y a su vez mediante la estructura, generar la nueva población. Este algoritmo esta basado en el presentado en articulo escrito por Miquélez et all. [23], llamado EBCOAs (Evolutionary Bayesian Classifier-based Optimization Algorithm) aunque en este caso está adaptado para un problema de valores discretos al contrario que el mencionado, que esta pensado para valores continuos.

Para poder implementar el TAN, se puede usar librerías de R como *bnlearn* o *bnclassify*, ya que ambas permiten utilizar redes bayesianas. En este trabajo se ha usado la librería bnlearn. El siguiente pseudocódigo explica de manera detallada la implementación utilizada y el código se puede ver en el anexo.

Algorithm 6 Método TAN + EDA

Require: $nIter, nPop, nTrunc, nAttr$ ▷ Hiper parámetros del algoritmo
  $Population \leftarrow generateInitialPopulation(nPop)$
  $i \leftarrow 0$
  while $i < nIter$ **do**
    $fitnessPopulation \leftarrow fitnessFunction(Population, nTrunc)$
    $fitnessPopulation \leftarrow dicotomizarItems(fitnessPopulation)$ ▷ Se dicotomiza la columna de los items
    if $lengthCategories(fitnessPopulation) == 1$ **then** ▷ Se comprueba si hay una sola clase en la población
      return ▷ Convergencia del método
    end if
    $Population \leftarrow genereneNewPopWithTAN(fitnessPopulation, nPop)$
    $i \leftarrow i + 1$
  end while

Siguiendo el pseudocódigo, dentro de la función *generateNewPopWithTAN*, que recibe como argumento la mejor población seleccionada por la función objetivo, se construye el modelo TAN y se genera la nueva población. Pero, como ya se ha mencionado, el TAN es un clasificador, por lo que antes se debe dicotomizar los resultados de la función objetivo en N clases que luego lo utilizará el TAN para crear el modelo. Si la función objetivo devuelve los items de cada base ordenadas de menor a mayor, se crean N clases atendiendo que contienen todo el conjunto de población. Por ejemplo, las bases que tengan más de 50 *items* y menos de 100, se consideran que pertenecen a la clase "base-normal", teniendo en cuenta siempre que la población con la que se crea el modelo debe de tener al menos dos clases (como mínimo debe de ser un problema de clasificador binario).

Además se debe intentar utilizar los mejores exponentes de cada clase y usar clases distantes, por ejemplo la mejor y la peor clase según los *items* que tengan, ignorando las clases intermedias. Esto hace que el modelo generado tenga mejor

calidad ya que simularía la variabilidad que se intenta buscar en la distribución que usan los EDAs. Esto puede ser complicado según la generación de elementos pudiendo hacer que el algoritmo converja a soluciones no óptimas. Por último, es necesario crean una condición de parada en caso de que el método converja antes que las iteraciones propuestas, siendo esto cuando la población generada solo pertenezca a una única clase.

3.2.3. Optimización de las soluciones propuestas

En este apartado se describen las mejoras de los métodos y funciones presentados con el objetivo de mejorar el rendimiento de los mismos. Para ello se estudia los cuellos de botella de los métodos de infraestructura, ya que son los métodos que se usan con mayor frecuencia en todos los algoritmos propuestos y son aquellos que manejan directamente el dataframe que contiene la tabla de decisiones óptimas.

Primero se observa que partes del código hacen que aumente el tiempo de ejecución y se estudian alternativas o mejoras. Esto se hace mediante el *profiler* que esta disponible en RStudio [27], entorno de desarrollo donde se ha codificado la solución y este software permite analizar el rendimiento del código.

El principal problema que se tiene con los métodos de infraestructura es que, al manejar el dataframe, están continuamente accediendo a la memoria que ocupa la estructura, haciendo que sean estas las partes más lentas del código. Esto es especialmente crítico cuando se trabaja con problemas de decisión grandes ya que su tabla de decisión asociada ocupa un gran tamaño en memoria, ocasionando que el cambio de columnas y el reordenamiento de elementos sea sustancialmente lento.

Es por ello se plantea en primer lugar cambiar el método de *swap column order*, función que cambia las columnas de orden y reordena los elementos de la nueva tabla según la primera columna. Inicialmente, para modificar las columnas, se guarda una copia de la columna origen para luego cambiar la columna destino. Esta aproximación, que se puede ver en el siguiente fragmento del código es ineficiente, especialmente en dataframes grandes.

Listing 3.2: Implementación inicial del cambio de columna

```
1  #Se cambia el contenido de las columnas
2  ret_df <- df
3  back <- ret_df[col_origen]
4  ret_df[col_origen] <- ret_df[col_destino]
5  ret_df[col_destino] <- back
6
7  #Se cambia el nombre de las columnas
8  back_name <- colnames(ret_df)[col_origen]
9  colnames(ret_df)[col_origen] <-colnames(ret_df)[col_destino]
10 colnames(ret_df)[col_destino] <- back_name
```

Para poder solucionar este cuello de botella (se recuerda que se esta cambiando constantemente de columna en todas las soluciones propuestas), se plantea esta solución, que evita crear una copia de la columna origen y que trabaja di-

rectamente con el dataframe original mediante los índices de esta, haciendo que vaya más rápido. Por otro lado, el ordenado de los elementos del dataframe no se puede mejorar de primeras, ya que se usa la función *order*, que ya es la opción más rápida de las que se dispone (hay otras como *rank()* o *sort()*).

Listing 3.3: Mejora del cambio de columna

```
#version optimizada
col_indices <- seq_along(df)
col_indices[c(col_origen, col_destino)] <- col_indices[c(col_destino, col_origen)]
ret_df <- df[, col_indices]
```

En cuanto al método *reduce*, el otro método de infraestructura, no es necesario modificarle ya que no es un punto crítico del código y además tampoco es oportuno (y posible de primeras) incluir paralelismo en el, ya que no habría ganancia o sería muy pequeña.

En cuanto al paralelismo se usa la librería doParallel junto con foreach, ambas de R, que permiten paralelizar bucles. Primero se define el número de procesadores que se va a usar, luego se crea el cluster con los procesadores y se registra, si fuese el caso, los ficheros o librerías que necesitarían los procesadores para correr la sección paralela. Después se define el foreach y como se devolvería, si procede, la información que se trabaja en el bucle. Finalmente, se destruye el cluster. A continuación se muestra un fragmento de código paralelizado, que se centra en la función *fitness*:

Listing 3.4: Paralelización del bucle que calcula los items

```
items_vector <- c(1:length_list)
 num_cores <- 6
 #Start the parallel backend
 cl <- makeCluster(num_cores)
 registerDoParallel(cl)
 clusterEvalQ(cl, {
   source("C:/Users/Enrique/.../calculate_kbm2l_from_base.R")
   source("C:/Users/Enrique/.../swap_columns_order.r")
   source("C:/Users/Enrique/.../reduce_kbm2l.R")
 })
 items_vector <-foreach(i = 1:length_list, .combine = c) %dopar% {
    calculate_kbm2l_from_base(c(list_population[i,]),base,dfx)
 }
 # Stop the parallel backend
 stopCluster(cl)
```

Por último, hay que mencionar que las estructuras principales con las que se han trabajado en R son las matrices, los dataframes y las listas. El rendimiento observado entre las dos primeras es similar y se podría trabajar con ambas, eso si, hay que evitar convertir las estructuras en otras, por ejemplo de dataframe a matriz o similares, ya que es una sobrecarga nada desdeñable, especialmente en los códigos descritos, que se ejecutan un gran número de veces.

3.3. Pruebas

En este apartado se describen el conjunto de pruebas con el que se ha evaluado el correcto funcionamiento del código programado. Debido al gran número de funciones, se ha dividido estas pruebas en dos grandes grupos, las pruebas de infraestructura y las pruebas de los métodos de optimización, que a su vez cubren las funciones implementadas en cada área.

3.3.1. Pruebas de infraestructura

Las pruebas de infraestructura se centran en el estudio del buen funcionamiento de las funciones de creación de la lista KBM2L (ver algoritmo 3) y el cambio de columnas (ver algoritmo 2) pertenecientes al apartado 3.2.1.3. A continuación, se presentan dos tablas que contienen las pruebas unitarias que se han utilizado para testear estas funciones (tablas 3.3 y 3.4):

Nº Prueba	Descripción	Input	Output
1	Prueba con índice mayor que la dimensión del dataframe	i=1,j = 6, dfx(10 filas, 5 columnas)	Error: índice erróneo
2	Prueba con índice negativo	i = -1, j = 1, dfx(10x5)	Error: índice erróneo
3	Prueba con índices iguales	i = j = 1, dfx(10x5)	Error: índices iguales
4	Prueba con índices correctos	i = 1, j = 2, dfx(10x5)	Columnas cambiadas y por orden de menor a mayor de la primera columna

Cuadro 3.3: Pruebas unitarias de *swap column order*

Nº Prueba	Descripción	Input	Output
1	Prueba con vector de 1x1	c(1)	Error: vector tamaño 1
2	Prueba con vector correcto 1	c(1,2,3,4,5) (1x5)	Matriz [[1,2,3,4,5],[0,1,2,3,4]] (5x2)
3	Prueba con vector correcto 2	c(1,1,2,2,3,3) (1x6)	Matriz [[1,2,3],[1,3,5]] (3x2)
4	Prueba con vector correcto 3	c(1,1,1,1,1) (1x5)	Matriz [[1],[4]]

Cuadro 3.4: Pruebas unitarias de *reduce*

En cuanto al input de la función *swap column order*, recibe los índices de las columnas que quieres hacer el cambio y el propio dataframe. Devuelve el mismo dataframe con las columnas cambiadas y los elementos ordenados por la primera columna de menor a mayor. Se prueba que los índices existen, que no

sean iguales y que no excedan las dimensiones del dataframe. Por otro lado, la función *reduce* recibe como argumento un vector (que es la columna de decisiones del dataframe) y devuelve una matriz con la KBM2L asociada (de n filas y 2 columnas). Se comprueba que no se pase un vector de 1 elemento.

Por último, para las dos funciones restantes de infraestructura (Diagramas de influencia a tablas de decisión y tablas de decisión a tablas de decisiones óptimas) se plantea el uso de problemas de decisión de prueba, en los que se puede comprobar el resultando mediante el software GeNIe, que realiza la misma tarea que el paquete IdR.

3.3.2. Pruebas de optimización

Las pruebas de optimización tratan de comprobar el funcionamiento del método UMDA, exhaustivo y TAN. Para ello se ha propuesto dos problemas de decisión para poder testear estos métodos. El primero, llamado *ictneo2HGM*, es un problema de decisión que cuenta con dos nodos decisores y que para hacerlo más pequeño se va a usar solo el primer nodo decisor (Terapia1). A su vez, el modelo usado es una versión reducida del planteado en [11], que tiene 5 nodos de decisión.

El segundo, al igual que el primero es una versión reducida del modelo llamado *NHLV2* [28], que cuenta para el trabajo con tres nodos decisores y se usará el modelo entero como prueba de estrés. Este modelo es una versión reducida del modelo desarrollado para el diagnóstico y tratamiento del linfoma. A continuación, se muestra la tabla 3.5 que contiene las dimensiones de los problemas de decisión (la tabla de decisiones óptima).

Modelo	Nº Filas	Nº Atributos	Nodos Decisión	Nº Decisiones
ictneo2HGM	432	6	1	4
nhvl2	2880	8	3	4

Cuadro 3.5: Características de los modelos utilizados en las pruebas

En cuanto al primer modelo, el problema de decisión ictneo2HGM [11] trata sobre la ictericia infantil y sobre las decisiones a tomar sobre el tratamiento mientras que el modelo NHLV2 [28] sobre la selección óptima del tratamiento contra el linfoma de Hodgkin (NHL), que es un tipo de cáncer. El modelo IctNeo (que se puede ver en la figura 3.1 el modelo completo) cuenta con 6 atributos, lo que se traduce en 720 bases disponibles (6!) mientras que el modelo NHLV2, que cuenta con 2 atributos más, pasa de 720 a 40320 bases (8!). En la figura 1, disponible en el anexo .1, se puede ver el diagrama de influencia (completo, el que se usa en el trabajo es una versión simplificada) del modelo NHL.

Capítulo 4

Resultados

En el siguiente capítulo se presentan los resultados obtenidos con los distintos métodos programados usando los dos modelos planteados en la sección 3.3.2. Se realiza un análisis de rendimiento junto con una comparación de resultados.

4.1. Análisis de rendimiento

Se presentan los resultados obtenidos en términos de rendimiento tales como el tiempo de ejecución y uso de memoria. Para ello se utilizan herramientas de benchmarks como la librería *microbenchmark* de R y el propio *profiler* que trae *RStudio*. Primero se muestra los resultados que arroja el profiler y luego se muestran los tiempos de ejecución obtenidos.

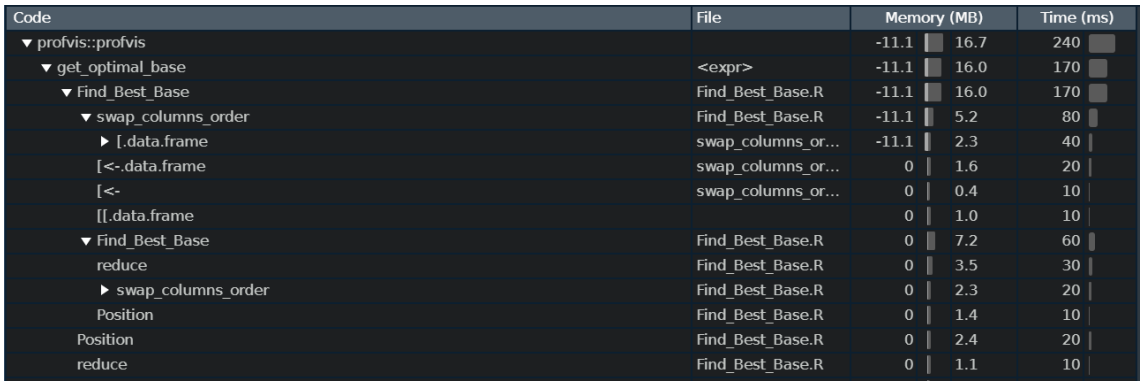
En cuanto a los tiempos de ejecución se utilizan dos problemas de decisión 3.3.2, una parte del problema del IctNeo (primera decisión de las dos que tiene) y el modelo NHLV2, que cuenta con un mayor tamaño, pudiendo comparar tiempos con un problema mas pequeño y uno más grande, que se denotarán a lo largo de este apartado como problema pequeño (que tiene dimensiones de 432 filas y 6 columnas) y problema grande para mayor simplicidad.

4.1.1. Análisis de rendimiento del método exhaustivo

Se recuerda que el método exhaustivo esta implementado de tal forma que escoge la mejor base del vecindario (aquella que tiene menos items que en la KBM2L asociada) y se vale de un método recursivo que va comprobando aquellas bases por las que no se ha visitado y calculando la KB2ML. Es importante mencionar que se hace una poda por aquellas bases por las que se ha visitado, algo muy frecuente en problemas de estructura de grafos. Esto permitirá aumentar la velocidad del algoritmo.

Se presenta los resultados del profiler para el problema pequeño. El profiler en RStudio presenta dos vistas, una que contiene un gráfico de llamadas a funciones presentes en la ejecución y lo presenta en forma de “stack”. La desventaja que tiene esta visualización es que en casos donde el método realiza muchas

llamadas (como este caso), su visualización es confusa. Para ello, se presenta la segunda visualización, que descompone en forma de árbol las llamadas de funciones involucradas y presenta su uso de memoria y tiempo de computo. En este caso, y debido a que para el problema pequeño el tiempo de ejecución es muy rápido y podría dar a lugar a resultados erróneos del profiler, se han realizado 20 iteraciones del mismo para ver los resultados que arroja la herramienta. Se pueden ver los resultados en la imagen 4.1.



Code	File	Memory (MB)		Time (ms)
▼ profvis::profvis		-11.1	16.7	240
▼ get_optimal_base	<expr>	-11.1	16.0	170
▼ Find_Best_Base	Find_Best_Base.R	-11.1	16.0	170
▼ swap_columns_order	Find_Best_Base.R	-11.1	5.2	80
▶ [.data.frame	swap_columns_or...	-11.1	2.3	40
[<-.data.frame	swap_columns_or...	0	1.6	20
[<-	swap_columns_or...	0	0.4	10
[[.data.frame		0	1.0	10
▼ Find_Best_Base	Find_Best_Base.R	0	7.2	60
reduce	Find_Best_Base.R	0	3.5	30
▶ swap_columns_order	Find_Best_Base.R	0	2.3	20
Position	Find_Best_Base.R	0	1.4	10
Position	Find_Best_Base.R	0	2.4	20
reduce	Find_Best_Base.R	0	1.1	10

Figura 4.1: Resultados profiler método exhaustivo

Comentando la imagen, vemos que la base óptima que arroja el método se obtiene tras dos llamadas a la función recursiva (se muestra información de las llamadas, uso de memoria y tiempos), esto es una llamada para obtener la siguiente base y la otra llamada en la que no se consigue una mejor base y se detiene. La base obtenida es siempre la primera mejor que se consigue, que para el caso del problema pequeño es la base $B_x = [3, 2, 1, 4, 5, 6]$ que tiene 25 items (recordamos que la inicial es la base $B_0 = [1, 2, 3, 4, 5, 6]$, por lo que la mejor base que se obtiene siguiendo este método es la vecina de la base inicial.

En cuanto al uso de memoria, que se muestra de tal forma que los números negativos muestran la memoria que se ha desasignado (deallocate) y el número positivo indica la memoria que se ha asignado (allocate), siempre en MegaBytes como unidad de medida. Vemos que dentro de la ejecución del código, lo que más consume del mismo es el cambio de base y el ordenamiento de los elementos del dataframe, todo ello perteneciente a la función *swap column order*. También se puede ver que la otra función de infraestructura (*reduce*) tiene poco impacto en el uso de memoria y en el tiempo de ejecución y que la segunda llamada a la función recursiva consume menos tiempo, debido a la poda y a que no se obtienen bases mejores.

En lo relativo al tiempo de ejecución, microbenchmark ofrece los siguientes resultados expresados en milisegundos (tabla 4.1) y se descompone en los tiempos mínimo, máximo, media, mediana y sus cuartiles.

Siguiendo la mejora en el uso de memoria en el método *swap column order* planteado en la sección 3.2.3, que emplea el método exhaustivo, se obtienen los siguientes tiempos, que se puede ver en la tabla.

Iteraciones	Mínimo	Cuartil Inferior	Media	Mediana	Cuartil Superior	Máximo
20	9.81	11.69	13.66	13.24	14.79	23.21

Cuadro 4.1: Tiempo de ejecución exhaustivo en problema pequeño

Iteraciones	Mínimo	Cuartil Inferior	Media	Mediana	Cuartil Superior	Máximo
20	7.40	8.13	10.53	9.47	11.00	20.80

Cuadro 4.2: Tiempo de ejecución exhaustivo en problema pequeño optimizado

Ahora, se muestran los resultados obtenidos en forma de tabla para el problema grande tanto en su versión normal como optimizada con 20 iteraciones. Al contrario que en el problema pequeño, el método ha encontrado la base óptima tras 13 iteraciones siendo esta base $B_x = [8, 6, 3, 4, 2, 5, 1, 7]$.

Versión	Mínimo	Cuartil Inferior	Media	Mediana	Cuartil Superior	Máximo
Normal	306.42	311.30	319.89	315.67	321.00	367.87
Optimizada	276.73	279.42	286.21	282.92	289.04	318.08

Cuadro 4.3: Tiempos de ejecución exhaustivo en problema grande en milisegundos

4.1.2. Análisis de rendimiento del método EDA UMDA

Siguiendo de forma análoga lo hecho en el apartado previo, se muestra los resultados que arroja el profiler, que se pueden ver en la imagen 4.2,con el método UMDA y se realiza un breve comentario.

El código del UMDA se divide en tres partes, la generación de la población en base a la distribución (*generate pop umda*), la evaluación de la población generada en la función fitness (*fitness function*) y el cálculo de la nueva distribución según las mejores muestras de la población objetivo (*update umda*). De estas tres funciones, la menos costosa tanto en términos de memoria y velocidad es la calcula la nueva distribución, ya que simplemente actualiza la matriz de frecuencias con los nuevos valores. Por el contrario, la más costosa es la que genera la nueva muestra, seguida muy de cerca de la función objetivo.

En el caso de la función primera, su coste computacional viene asociado a que por un lado, se están creando de cero las bases (se usa de manera intensiva la función *order*) y al igual que la función *runif*, que genera números aleatorios. Para la segunda función, la complejidad computacional viene dada por el manejo de los dataframes (mencionado en el capitulo 3.2.3) y también por el uso de la función *order*. Se muestran los tiempos asociados (en segundos) usando 20 iteraciones del problema pequeño en la tabla 4.4. En cuanto a la prueba usando

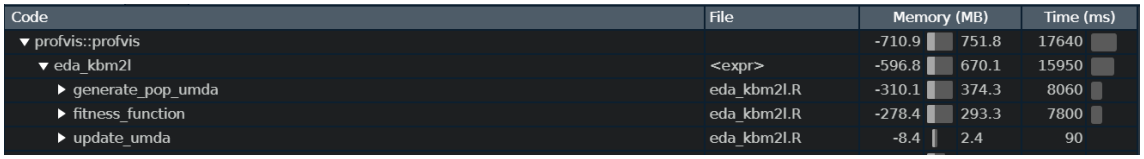


Figura 4.2: Resultados profiler método UMDA

paralelismo, se ha usado 6 núcleos del procesador.

Versión	Mínimo	Cuartil Inferior	Media	Mediana	Cuartil Superior	Máximo
Normal	22.67	25.06	29.41	27.96	30.17	53.94
Paralelismo	215.37	222.39	225.36	224.70	226.63	243.93
Memoria	9.51	10.63	12.61	11.25	12.71	35.31

Cuadro 4.4: Tiempos de ejecución en segundos del EDA en problema pequeño

A continuación se muestra en la siguiente tabla los resultados obtenidos con el método EDA en el problema grande (20 iteraciones y datos en segundos). También se puede ver los tiempos de la versión optimizada con paralelismo y la optimización de memoria.

Versión	Mínimo	Cuartil Inferior	Media	Mediana	Cuartil Superior	Máximo
Normal	24.33	26.23	31.20	27.39	30.08	80.99
Optimizado	23.99	25.81	29.77	27.43	28.53	72.97

Cuadro 4.5: Tiempos de ejecución EDA en problema grande en segundos

4.1.3. Análisis de rendimiento del TAN

De forma análoga que en las secciones previas, se muestra el resultado obtenido por el profiler figura 4.3) y se muestran una tabla con los tiempos obtenidos tras 20 iteraciones del código tanto con el problema pequeño como con el grande en sus formas normal y optimizada, expresados en segundos.

La imagen muestra de nuevo que la función fitness u objetivo es la que más consume en términos de memoria y de tiempo, siendo la función que genera la nueva población y que usa el clasificador TAN muy rápida en términos de velocidad y con poco uso de la memoria. La tabla 4.6 muestra los tiempos en milisegundos del modelo IctNeo tanto en su versión normal como optimizada.

En la tabla 4.7, se puede ver los tiempos en milisegundos del TAN en el modelo NHVL2 tanto en su versión normal como la optimizada.

Code	File	Memory (MB)	Time (ms)
▼ profvis::profvis		-5.8 9.9	160
▼ eda_tan_kbm2l	<expr>	-5.8 8.5	130
▶ fitness_function_tan	eda_kbm2l.R	-5.8 7.1	110
▶ generate_new_pop_tan	eda_kbm2l.R	0 1.4	20
▶ Rprof		0 0	10
as.character		0 0.8	10
invisible		0 0.5	10

Figura 4.3: Resultados profiler método TAN

Versión	Mínimo	Cuartil Inferior	Media	Mediana	Cuartil Superior	Máximo
Normal	87.11	117.96	129.64	121.87	130.50	226.84
Optimizada	39.16	73.18	86.09	79.35	105.94	126.59

Cuadro 4.6: Tiempos de ejecución TAN en problema pequeño en milisegundos

4.2. Comparación de resultados

Se puede notar que se han obtenido resultados distintos entre los diferentes métodos. Tanto en el problema pequeño como en el problema grande, el ganador es el método exhaustivo, ya que es el que obtiene mejores tiempos. La imagen 4.4 muestra una comparativa de los tiempos (en base 10) en milisegundos de los diferentes métodos en el problema pequeño, mientras que la imagen 4.5 hace lo mismo para el problema grande (en orden normal-optimizado).

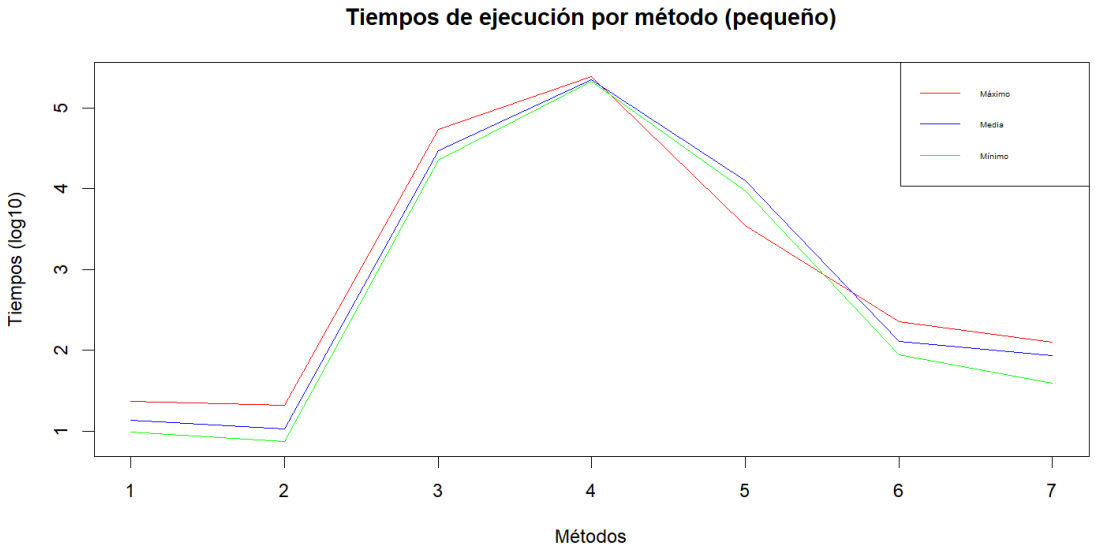


Figura 4.4: Gráfico con tiempos (milisegundos) en problema pequeño (1-2 Exhaustivo, 3-5 UMDA, 6-7,TAN)

Esto se puede explicar debido a que, para el caso del problema pequeño, de primeras ya necesita calcular menos bases que la generación inicial de cualquier método EDA o TAN para obtener la lista óptima. Y esto es debido a la poda

Versión	Mínimo	Cuartil Inferior	Media	Mediana	Cuartil Superior	Máximo
Normal	171.15	176.19	184.32	179.52	188.41	223.23
Optimizada	156.19	159.78	166.77	163.90	169.63	203.73

Cuadro 4.7: Tiempos de ejecución TAN en problema grande en milisegundos

que se realiza al no visitar bases ya calculadas previamente, algo muy común especialmente en iteraciones ya avanzadas del método y que le permiten avanzar o parar aún más rápido.

Sin embargo el método tiene dos desventajas claras y es que por un lado escala de forma notoria entre los problemas debido a que es un método exhaustivo (obsérvese que el aumento de tiempo entre el problema pequeño y grande es mayor que para el resto de métodos) y por otro lado, no garantiza obtener el óptimo absoluto ya que se puede quedar en un óptimo local (se hace necesario en determinados casos empezar en distintas semillas, esto es, distintas generaciones de población inicial).

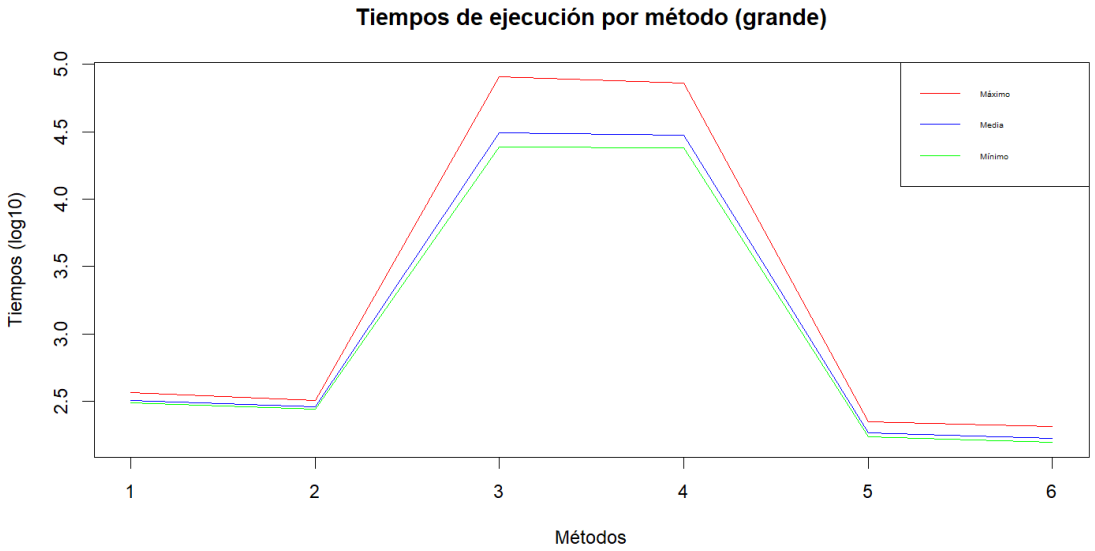


Figura 4.5: Gráfico con tiempos (milisegundos) en problema grande (1-2 Exhaustivo, 3-4 UMDA, 5-6 TAN)

En cuanto al método EDA UMDA se pueden extraer las siguientes conclusiones.

- Método estable en cuanto al consumo computacional, no hay un gran salto de tiempo entre el problema pequeño y el problema grande (frente al exhaustivo) pero tiene unos cuantos puntos negativos.
- El primero es que se necesita ajustar apropiadamente los parámetros de selección y generación de población en casos donde el problema no sea pequeño.

- Lo segundo y relacionado con el primero, es que la muestra inicial es importante para que el método converja a una solución aceptable. Esta debe de ser variada y con pocas muestras que tengan el mismo valor en la función objetivo. Hay que recordar que el UMDA es el EDA más sencillo y tiene una hipótesis de independencia entre las variables.
- Por otro lado, los resultados utilizando el paralelismo son decepcionantes. Esto tiene una razón y es que la función objeyivo, que es un método que consume muchos recursos, tanto computacionalmente pero sobre todo de memoria. Y es que maneja constantemente la base de cocimiento del modelo, luego crea dependencias entre los hilos haciendo que sea incluso peor que la versión original, debido a los interbloqueos que suceden.

Si pasamos al método TAN, se puede ver que se obtienen mejores resultados que el método UMDA. Sin embargo, los puntos negativos que tiene el UMDA también los tiene el TAN pero se le añaden las siguientes consideraciones:

- La primera es que se hace necesario programar una condición de salida del algoritmo cuando se converja antes que las iteraciones propuestas. La librería *bnlearn* funciona extremadamente rápido, pero necesita que la base de conocimiento este codificada como factor (coste computacional de la conversión).
- Otra cuestión a tener en cuenta es que se necesita, para el clasificador, que los resultados de la función objetivo estén agrupados por clases, esto se puede hacer de varias formas, pero resulta un aspecto esencial en el buen funcionamiento del método ya que de su definición depende de que se converja a una solución buena o no. En el caso del trabajo se ha optado por una agrupación manual, aunque se puede utilizar la distribución obtenida de *items* para generar las clases.
- Por último, es vital programar una función que restaure las permutaciones, esto es modificar la población generada para que cumpla que cada elemento de la base sea único, algo que no se suele cumplir en la generación de muestras del TAN.

Hay que hacer especial mención a la generación de la población inicial, ya que es clave en los métodos EDAs. Se proponen varias lineas en este sentido en el apartado 5.3.

La clave en este aspecto es que la población sea lo más variada posible, sin que haya muchos elementos con el mismo resultado en la función objetivo, tal y como se menciona en el articulo de E. Bengoetxea et al [29]. Si esto ocurre, lo más probable es que el método converja a esas soluciones, que pueden ser buenas o malas (lo más probable es que sea lo segundo ya que el población de soluciones buenas suele ser menor).

En lo relativo a la explicabilidad, se puede observar que paralelamente a la búsqueda de la mejor base, también se están ordenando los atributos según su importancia. En la imagen 4.6, se puede ver una de las generaciones iniciales de la población usando el EDA junto con el TAN del modelo IctNeo (pequeño) [11].

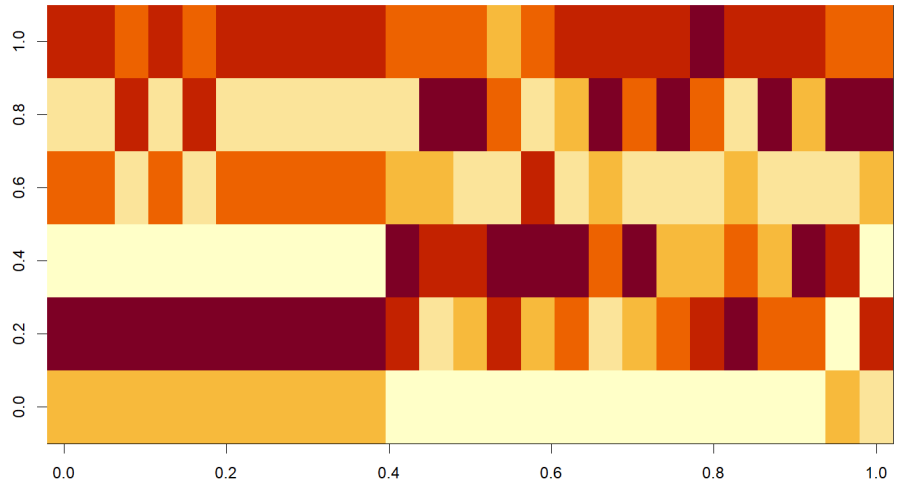


Figura 4.6: Espectro inicial de la población generada por el TAN

En ella, (recordamos que las generaciones iniciales son aleatorias), ya hay presentes varios candidatos a óptimos (25 elementos seleccionados tras la función objetivo), que son los que suelen tener el atributo 3 (Riesgo Ingreso) como atributo de mayor peso en la base. Los atributos situados a la derecha de la imagen son los que están más fragmentados, ya que cada muestra lo tiene ordenado de forma distinta. A continuación, se muestra la generación final (figura 4.7) del método TAN en el mismo problema.

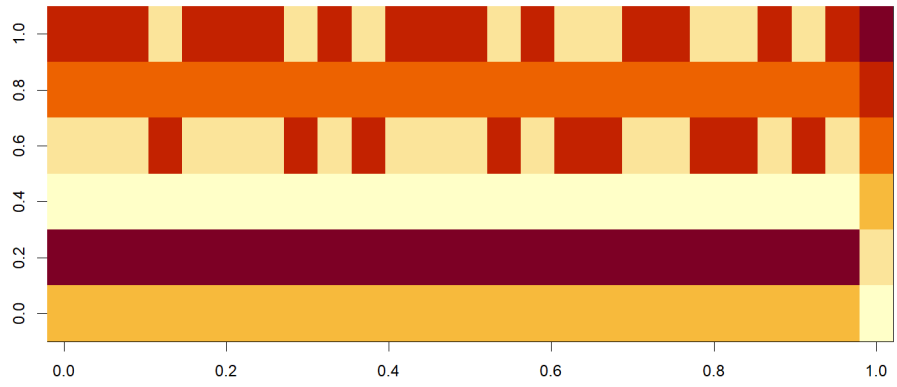


Figura 4.7: Espectro final de la población generada por el TAN

Podemos ver que en la generación final la mayoría de columnas ya siguen un patrón, exceptuando los últimos elementos de la población, y la cual casi todos

los miembros de la misma son soluciones óptimas. De forma implícita se esta dando un orden de atributos según su importancia y aquellos atributos que van variando según la muestra, pero cuya base sigue siendo óptima, son atributos poco relevantes para la decisión.

Capítulo 5

Conclusiones, Líneas Futuras y Análisis de Impacto

En este capítulo se enumeran las conclusiones extraídas del trabajo y se realiza una evaluación objetivos cumplidos del mismo junto con las líneas a trabajar en futuros trabajos relacionados y una evaluación del proceso de realización del TFM.

5.1. Conclusiones

Se pueden extraer varias conclusiones de este trabajo. La primera es la confirmación de la dificultad a la hora de trabajar con problemas de decisión de tamaño mediano y grande. Las tablas de decisión escalan según se aumenta el tamaño del problema (el número de atributos), haciendo indispensable el uso de técnicas alternativas para obtener y evaluar los resultados. La solución de utilizar las listas KBM2L reducen efectivamente la dimensionalidad de la tabla, siendo muy eficiente la conversión de las tablas de decisiones óptimas a listas KBM2L. Sin embargo, para obtener la lista KBM2L óptima se nos presenta un problema combinatorio de complejidad NP-Completo, que si bien es cierto que se puede atajar de forma efectiva con métodos "voraces"tradicionales, resulta inviable cuando se va escalando el problema.

Es por ello que se plantea el uso de métodos como los algoritmos genéticos y, en el caso de este trabajo, el uso de los EDAs, ya que tienen un complejidad fija, al contrario que los métodos tradicionales, que escalan según el tamaño del modelo. Ahora bien, hay que estudiar detenidamente el problema antes de elegir que tipo de EDA usar, de forma similar a lo que se hace con los algoritmos de aprendizaje automático. El caso del UMDA es paradigmático, ya que es muy sencillo, pero en determinados problemas se obtienen resultados decepcionantes. No obstante, a medida que se escogen EDAs más potentes, aumenta de forma paralela la complejidad a la hora de programar.

Sin embargo el uso de las KBM2L trae un efecto secundario muy positivo y es que a la hora de obtener la lista KBM2L óptima, se puede observar también que

atributos del problema son los más importantes a la hora de tomar la decisión, ya que los atributos que están situados más a la izquierda en la tabla, es decir, en puestos con mayor peso a la hora de construir la lista, hacen que que el tamaño de la lista generada sea menor. Esto permite obtener nuevo conocimiento sobre el problema de decisión y es información muy útil tanto para el programador como para el experto y personas interesadas en el problema. Por ejemplo, podría ayudar a validar que el modelo está bien construido.

Siguiendo con la validación del problema de decisión, surgen una serie de preguntas, ¿los problemas de decisión con un gran numero de atributos son realistas? ¿Los humanos necesitamos, a la hora de tomar decisiones, considerar un gran número de características?¿El que haya un gran número de atributos no hace sino aumentar la indecisión? Son preguntas interesantes y abiertas, y que cada persona tendrá su propia opinión. Es interesante hacer notar esto debido a que a mayor número de atributos mayor es la complejidad del problema y es que a veces, cuando un problema es grande, es posible sintetizar los atributos presentes en otros con el objetivo de hacer más sencillo la decisión.

En relación con los EDAs (y el TAN), es imprescindible en problemas que no sean pequeños, escoger de manera inteligente la población inicial, que es la que guiará al algoritmo a la solución óptima. Y es que sino se hace esto es muy probable que los métodos converjan a soluciones malas. Por otro lado, en problemas pequeños, es posible que inicialmente se obtengan soluciones prometedoras, haciendo que los métodos viren de forma correcta a las soluciones óptimas. Otro consideración a tener en cuenta, mencionado en el trabajo de E. Bengoetxea, et all [29], es que los EDAs tienen problemas al trabajar con redundancia de los individuos. Esto es población distinta con la mismo resultado en la función objetivo, pudiendo ocasionar problemas en la búsqueda de la mejor solución.

Finalmente destacar, en el ámbito de la programación, el cuello de botella que es el trabajar con el dataframe (es decir, la tabla de decisión). Las funciones que trabajan directamente con ella son las que más consumen computacionalmente pero a la vez son las que son más difíciles son de optimizar y paralelizar, ya que hay dependencias de memoria. Es indispensable realizar el menor número de accesos al dataframe y que estos estén lo mejor optimizados posibles.

5.2. Evaluación de los objetivos

Se plantearon varios objetivos en la introducción que se pueden ver a continuación:

1. El concepto de los Sistemas de Soporte de Toma de Decisiones junto con sus representaciones matemáticas, aplicaciones en el mundo real y librerías asociadas.

2. Las técnicas de optimización, desarrolladas en R, para poder construir una KMB2L partiendo de una base de conocimiento, todo ello construido en un paquete en R.

3. Una forma de poder comunicar con el usuario las decisiones que toma el modelo y las razones por las que se basa esa decisión.
4. Pruebas del código realizado.

De ellos, se cumplen totalmente el primer, tercero y cuarto punto, siendo el segundo cumplido parcialmente, ya que solo quedaría por crear la librería que recoja todo el código programado. Por otro lado, en lo relativo al tercer apartado, sería una linea futura escribir una función que ordene las características o atributos de mayor a menor peso o un procedimiento de consulta de ellas. Todo esto se menciona en las líneas futuras.

5.3. Líneas futuras

Este trabajo es solo un esbozo de las diferentes técnicas que se pueden aplicar y es que se pueden seguir varias líneas futuras relacionadas con la optimización del problema. Se puede dividir como dos grandes líneas futuras: estudio de los diversos EDAs en distintos problemas, que involucren o no la interacción de las variables del modelo y mejoras en los métodos de computación de alto rendimiento.

En lo relativo al primero, se pueden seguir implementando diversos EDAs, ya que se recuerda que el UMDA pertenece a los del tipo univariante. Se podría hacer un estudio analítico en distintos problemas de decisión comparando los distintos problemas de decisión y su rendimiento. También es interesante probar varios EDAs con diferentes muestras iniciales o semillas. Para este trabajo, se diseño un prototipo de EDA que utilizaba una distribución normal como generador de población en vez de una distribución de frecuencias que usaba el UMDA. Sin embargo para el caso de la distribución normal era necesario una restauración de la permutación obtenida.

Para mejorar el rendimiento de los EDAs se puede crear una lista de bases ya calculadas, que permitan almacenar el resultado de la función fitness y que cuando se genere una base ya calculada no se tenga que volver a calcular. Del mismo modo, se puede crear una función que elimine bases repetidas, algo muy común en casos donde el problema sea pequeño.

Siguiendo con la restauración de las permutaciones, para el trabajo se planteo como una función sencilla que simplemente cambiaba las columnas repetidas con las columnas no usadas. Esto puede ocasionar que se pierda parte de la estructura de la muestra que se ha generado, luego se puede proponer funciones de permutación más complejas, por ejemplo, una que use el conjunto de muestras para cambiar las columnas repetidas y no solo de muestra en muestra, haciendo que se respete la estructura de la muestra.

La otra línea futura son los métodos de computación de alto rendimiento. Se ha mencionado que el principal cuello de botella es el manejo del dataframe, ocasionando que cualquier intento de paralelizar esa parte ocasione interbloqueos. Se plantea que cada hilo que se lance tenga una parte del dataframe original y que tras varias iteraciones del algoritmo, haya un punto de encuentro. Esto permite

que no ocurran interbloqueos al acceder al dataframe, pero también obliga a que haya permutaciones prohibidas, ya que cada hilo solo tiene información parcial del modelo. Además, no podría haber muchos hilos ejecutando el algoritmo, ya que la partición de la tabla sería tan pequeña que habría muchas restricciones a la hora de generar bases.

Asimismo, para este trabajo se planteo inicialmente que la parte del código más demandante se ejecutaría en lenguaje C. Existen varios métodos para hacer que una función escrita en C se ejecute en lenguaje R, la más sencilla es con la función .C de R, que no es más que un "wrapper" a la función escrita en C, que debe de pertenecer a una librería dinámica enlazada en el lenguaje en R. Además, al estar escrito en C, permite el uso de librerías de computación de alto rendimiento como *OpenMP* [30], que permite lanzar hilos ligeros. Se añade que C, por el manejo de punteros, permite que el código vaya más rápido que su versión de R, aunque también es cierto que el compilador de R y muchas librerías de R están muy optimizadas.

Por último, algo ya mencionado en el apartado anterior, y es que se podría escribir una función que ordene de mayor a menor el peso que tienen los atributos a la hora de tomar la decisión. Es algo que ya viene de implícito, gracias a las listas KBM2L, pero es una capa de abstracción más de cara a facilitar al decisor la toma de decisiones o a la hora de verificar el modelo. Todo el código programado se puede recoger en un paquete en R.

5.4. Evaluación del proceso de realización del TFM

El trabajo se ha realizado mediante la supervisión constante del tutor utilizando reuniones regulares en los que se han discutido dudas, consideraciones relativas al trabajo... La memoria se ha ido revisando usando sucesivos borradores.

5.5. Análisis de Impacto

Como se ha descrito, los DSS están en auge y tienen un gran número de aplicaciones en distintos ámbitos. El que se desarrollen técnicas para reducir el consumo a nivel de memoria y de rendimiento podría hacer que se reduzca el impacto medioambiental, ya que, a mayores necesidades computacionales, mayor es el consumo energético necesario para correr los programas. Por otro lado a nivel empresarial puede ayudar a hacer que se tomen decisiones más rápido y con menor riesgo asociado. Esto permitirá a las empresas ahorrar dinero debido a que se minimizan los errores además de poder utilizar el tiempo en otro tipo de tareas. Por último, mencionar el impacto que tienen los DSS en conocer más a fondo los problemas que tratan y su capacidad de poder enseñar a las personas nuevos conocimientos.

Este trabajo se podría asociar entonces con los siguientes objetivos de desarrollo sostenible (ODS):

- Energía asequible y no contaminante (ODS 7), debido a que el trabajo trata de aumentar la eficiencia y la sostenibilidad energética de los DSS.
- Producción y consumo responsable (ODS 12), ya que los DSS pueden ayudar a aumentar la eficiencia de la toma de decisiones logrando la gestión sostenible de empresas tanto públicas como privadas.

Bibliografía

[1] L. Starita. (2021) Would you let artificial intelligence make your pay decisions? [Online]. Available: https://www.gartner.com/smarterwithgartner/would-you-let-artificial-intelligence-make-your-pay-decisions

[2] M. G. Juan Antonio Fernandez del Pozo, Concha Bielza, "A list-based compact representation for large decision tables management," *European Journal of Operational Research 160 (2005) 638–662*, 2003.

[3] D. J. Power. (2007) A brief history of decision support systems. [Online]. Available: https://dssresources.com/history/dsshistory.html

[4] S. R. Insua, C. B. Lozoya, and A. M. Caballero, *Fundamentos de los sistemas de ayuda a la decisión*. RA-MA, 2001.

[5] Barsh, Norman T. (Accessed 2023) Operations Research: An Introduction - Part IX. https://home.ubalt.edu/ntsbarsh/opre640a/partIX.htm.

[6] M. J. E. Howard R. A., "Influence diagrams. decision analysis," *American Psychological Association*, 2005. [Online]. Available: https://pubsonline.informs.org/doi/10.1287/deca.1050.0020

[7] B. Kamiński, M. Jakubczyk, and P. Szufel, *A framework for sensitivity analysis of decision trees*. Central European journal of operations research, 2018.

[8] F. Mata-Toledo, "TreePlan: Decision Tree Add-in for Excel," https://www4.ujaen.es/~fmata/sitd/tree164x.pdf, Accessed 2023.

[9] J. V. Neumann and O. Morgenstern, *Theory of Games and Economic Behavior*. Princeton University, 1944.

[10] BayesFusion, "BayesFusion," https://www.bayesfusion.com/genie/, Accessed 2023.

[11] G. M., "A graphical decision-theoretic model for neonatal jaundice," *Medical Decision Making*, 2007.

[12] S. M. Lundberg, G. Erion, H. Chen, A. DeGrave, J. M. Prutkin, B. Nair, R. Katz, J. Himmelfarb, N. Bansal, and S.-I. Lee, "From local explanations to global understanding with explainable ai for trees," *Nature Machine Intelligence*, vol. 2, no. 1, pp. 2522–5839, 2020.

[13] A. Altmann, L. Tolosi, O. Sander, and T. Lengauer, "Permutation importance: A corrected feature importance measure," *Bioinformatics (Oxford, England)*, vol. 26, pp. 1340–7, 04 2010.

[14] D. Gunning, E. Vorm, Y. Wang, and M. Turek, "DARPA's explainable AI (XAI) program: A retrospective," nov 2021. [Online]. Available: https://doi.org/10.22541%2Fau.163699841.19031727%2Fv1

[15] K. P. Murphy, *Machine Learning: A Probabilistic Perspective (Adaptive Computation and Machine Learning series)*. The MIT Press, 2012.

[16] C. Molnar, *Interpretable Machine Learning*, 2nd ed. online, 2022. [Online]. Available: https://christophm.github.io/interpretable-ml-book

[17] G. Fick and R. H. Sprague, *Decision Support Systems: Issues and Challenges*. International Institute for Applied Systems Analysis (IIASA), 2013.

[18] P. G. Muhlenbein H., "From recombination of genes to the estimation of distributions i. binary parameters," *Proceedings of the 4th International Conference on Parallel Problem Solving from Nature*, 1996.

[19] H. J.H, *Genetic Algorithms*. Scientific American, 1992. [Online]. Available: http://dx.doi.org/10.1038/scientificamerican0792-66

[20] R. Armañanzas, I. Inza, R. Santana, Y. Saeys, J. L. Flores, J. A. Lozano, Y. V. de Peer, R. Blanco, V. Robles, C. Bielza, and P. Larrañaga, "A review of estimation of distribution algorithms in bioinformatics," *BioMed Central*, 2008. [Online]. Available: http://www.biodatamining.org/content/1/1/6

[21] H. M. Pedro Larrañaga, José A. Lozano, "Estimation of distribution algorithms applied to combinatorial optimization problems," *Inteligencia Artificial. Revista Iberoamericana de Inteligencia Artificial*, 2003. [Online]. Available: https://www.redalyc.org/pdf/925/92571910.pdf

[22] N. Friedman, D. Geiger, and M. Goldszmidt, "Bayesian network classifiers. machine learning," *APA*, 1997.

[23] T. Miquélez, E. Bengoetxea, A. Mendiburu, and P. Larrañaga, "Combining bayesian classifiers and estimation of distribution algorithms for optimization in continuous domains," *Connection Science*, 2007. [Online]. Available: https://doi.org/10.1080/09540090701725524

[24] M. Jones. (2017) Parallel Computing in R. https://nceas.github.io/oss-lessons/parallel-computing-in-r/parallel-computing-in-r.html.

[25] C. B. Juan Antonio Fernandez del Pozo, "Influence diagrams on r," *UPM*, 2007. [Online]. Available: http://www.dia.fi.upm.es/~jafernan/research/idr/Submission-idr.pdf

[26] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische mathematik*, vol. 1, no. 1, pp. 269–271, 1959.

[27] RStudio Team, *RStudio: Integrated Development Environment for R*, RStudio, PBC., Boston, MA, 2020. [Online]. Available: http://www.rstudio.com/

[28] L. P. Bielza C., Fdez del Pozo J.A., "Explanation of clinical decisions through the extraction of regularity patterns," *ScienceDirect*, 2008. [Online]. Available: https://doi.org/10.1016/j.dss.2007.05.002

[29] E. Bengoetxea, P. Larrañaga, C. Bielza, and J. A. F. del Pozo, "Optimal row and column ordering to improve table interpretation using estimation of distribution algorithms," *Journal of Heuristics*, 2010.

[30] OpenMP Architecture Review Board, "OpenMP application program interface version 3.0," May 2008. [Online]. Available: http://www.openmp.org/mp-documents/spec30.pdf

[31] Enrique Jiménez Fernández, "TFM_SETDO: Título del repositorio," https://github.com/Enricjfz/TFM_SETDO, 2023.

Anexos

Anexo

Debido a la extensión del mismo, se muestran algunas partes del código desarrollado y el diagrama de influencia del modelo NHLV2. El resto del código se puede encontrar en el repositorio de github [31]. El anexo se divide al igual que los apartados del desarrollo, primero, el código de infraestructura y luego los métodos heurísticos.

.1. Diagrama de influencia NHLV2

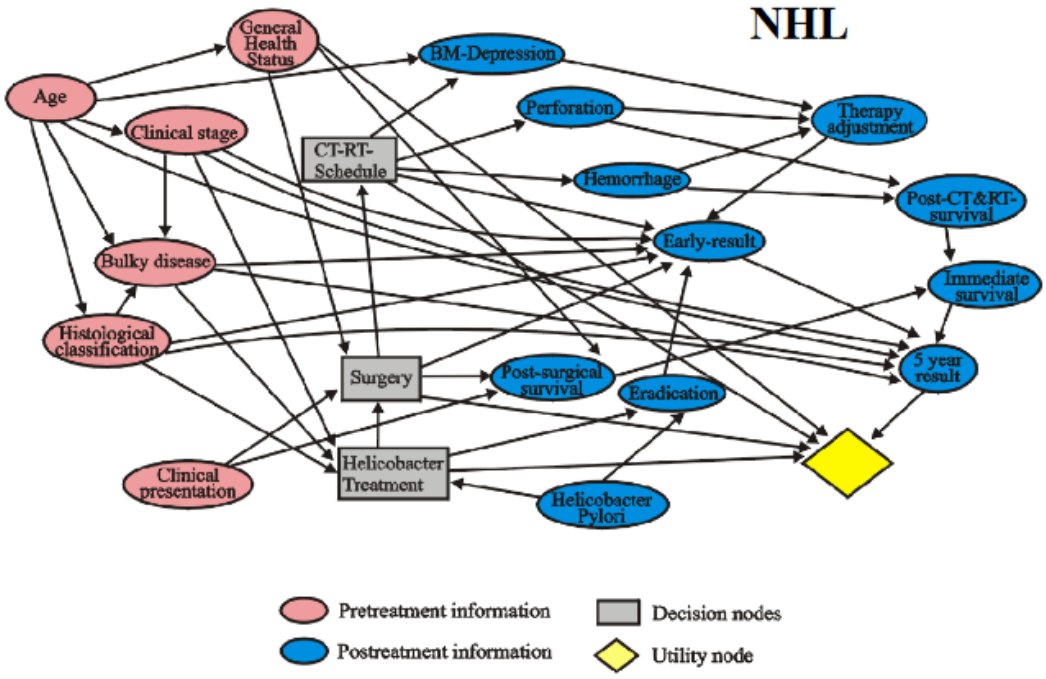


Figura 1: Diagrama de influencia del modelo NHLV2

.2. Código de infraestructura

Código de infraestructura desarrollado para el manejo de bases.

.2.1. Función Swap columns order

Listing 1: Función swap columns order

```
swap_columns_order <- function(df, col_origen, col_destino, order) {
  n_cols = dim(df)[2]
  if(col_origen > n_cols || col_origen < 1 || col_destino < 1 || col_destino > n_cols ||
       col_origen == col_destino) {
    cat("Wrong indexes\n")
    return (-1)
  }

 #Se cambia el contenido de las columnas
 ret_df <- df
 back <- ret_df[[col_origen]]
 ret_df[[col_origen]] <- ret_df[[col_destino]]
 ret_df[[col_destino]] <- back

 #Se cambia el nombre de las columnas
 back_name <- colnames(ret_df)[col_origen]
 colnames(ret_df)[col_origen] <- colnames(ret_df)[col_destino]
 colnames(ret_df)[col_destino] <- back_name

 #version optimizada
 #col_indices <- seq_along(df)
 #col_indices[c(col_origen, col_destino)] <- col_indices[c(col_destino, col_origen)]
 #ret_df <- df[, col_indices]

 ##Se ordenan las columnas segun la primera de estas si order es true

 if(order) {
   ret_df <- ret_df[order(ret_df[[1]]),]
 }

  return(ret_df)
}
```

.2.2. Función Reduce

Listing 2: Función reduce

```
reduce <- function(arr) {
  n <- length(arr)

  if (n <= 2) {
    return(arr)  # No se puede reducir mas el vector
  }

  idx <- 1  # Indice actual en el vector reducido
  reduced_arr <- c(arr[1])  # Vector reducido
  offset_arr <- c() #Vector de offset (empieza en 1)


  for (i in 2:n) {
    # Si el elemento actual es distinto de sus vecinos, anadirlo al vector reducido
    if (arr[i] != arr[i - 1]) {
      #cat("offset : ",i-1,"\n")
      offset_arr[idx] <- (i-2)
      reduced_arr[idx] <- arr[i-1]
      idx <- idx + 1
    }
  }

  reduced_arr <- c(reduced_arr, arr[n])  # Anadir el ultimo elemento
```

```
  offset_arr <- c(offset_arr,n-1) # Ultimo offset
  result <- cbind(reduced_arr,offset_arr)

  return(result)
}
```

.2.3. Función Get Optimal Table

Listing 3: Función Get Optimal Table

```
reduce <- function(arr) {
  n <- length(arr)

  if (n <= 2) {
    return(arr)  # No se puede reducir mas el vector
  }

  idx <- 1  # Indice actual en el vector reducido
  reduced_arr <- c(arr[1])  # Vector reducido
  offset_arr <- c() #Vector de offset (empieza en 1)


  for (i in 2:n) {
    # Si el elemento actual es distinto de sus vecinos, anadirlo al vector reducido
    if (arr[i] != arr[i - 1]) {
      #cat("offset : ",i-1,"\n")
      offset_arr[idx] <- (i-2)
      reduced_arr[idx] <- arr[i-1]
      idx <- idx + 1
    }
  }

  reduced_arr <- c(reduced_arr, arr[n])  # Anadir el ultimo elemento
  offset_arr <- c(offset_arr,n-1) # Ultimo offset
  result <- cbind(reduced_arr,offset_arr)

  return(result)
}
```

.3. Métodos heurísticos

.3.1. Método exhaustivo

Listing 4: Función Exhaustiva (método recursivo)

```
get_optimal_base <- function(dfx, base_inicial) {
  #iniciacion del metodo recursivo
  attr <- length(base_inicial)
  if(dim(dfx)[2]-1 != attr) {
    cat("wrong initial base\n")
    return(-1)
  }

  lista_bases_visitadas <- list() #inicializacion de la lista
  indice_lista_actual <- 1
  lista_bases_visitadas[[indice_lista_actual]] <- base_inicial
  indice_lista_actual <- indice_lista_actual + 1
  best_perm <- Find_Best_Base(dfx,base_inicial,lista_bases_visitadas,indice_lista_actual
      )
  return(best_perm)
```

```
15
16 }
17
18 Find_Best_Base <- function(dfx, base_actual,lista_bases, index_lista) {
19
20   n_col <- dim(dfx)[2] -1 #no usamos la columna de decisiones
21   p_perm <- base_actual #posible permutacion
22   best_perm <- base_actual #mejor permutacion
23   dfx_inicial <- dfx #kb original
24   best_dfx <- dfx
25   stop_condition <- 0
26
27   kbm2l_actual <- reduce(dfx[[n_col+1]]) #obtenemos la kmb2l inicial
28
29   for (i in 1:n_col) {
30     for(j in 1:n_col) {
31       if(i == j) {
32         next #no permutation
33       }
34       #swap base, se comprueba si esta en la lista
35       back <- p_perm[i]
36       p_perm[i] <- p_perm[j]
37       p_perm[j] <- back
38
39       if (Position(function(x) identical(x, p_perm), lista_bases, nomatch = 0) > 0) {
40         #esta en la lista, se salta iteracion
41         p_perm <- base_actual
42         next
43       }
44
45       lista_bases[[index_lista]] <- p_perm
46       index_lista <- index_lista + 1
47       dfx_swapped <- swap_columns_order(dfx,i,j,1)
48       new_kbm2l <- reduce(dfx_swapped[[n_col +1]])
49       if(dim(new_kbm2l)[1] < dim(kbm2l_actual)[1]) {
50         #hemos encontrado una base mejor
51         print(dim(new_kbm2l)[1])
52         print(p_perm)
53         best_dfx <- dfx_swapped
54         kbm2l_actual <- new_kbm2l
55         best_perm <- p_perm
56         stop_condition <- 1
57       }
58
59       p_perm <- base_actual #volvemos a la base inicial
60
61
62     }
63
64   }
65   if(stop_condition) {
66     #hemos encontrado una base mejor, se sigue iterando
67     return(Find_Best_Base(best_dfx,best_perm,lista_bases,index_lista))
68
69   }
70   else {
71     #hemos encontrado una base mejor, supuesto optimo
72     return(best_dfx)
73   }
74
75 }
```

.3.2. Método EDA UMDA

Listing 5: Método EDA UMDA

```
eda_kbm2l <- function(dfx,base_inicial,iter,pop,sel) {
  #Function which uses the EDA algorithm to calculate the optimal base
  #dfx is the initial dataframe of the decision problem
  #base_inicial is the initial base
  #iter is the number of iterations of the problem
  #pop is the population to be generated
  #sel is the selection of the population after the fitness


  attr <- length(base_inicial)
  #initial_pop <- generate_initial_population(pop,attr)
  #new_pop <- initial_pop
  M <- calculate_initial_univariate_distribution(attr)
  for(i in 1:iter) {
    new_pop <- generate_pop_umda(M,pop,attr)
    fitness_pop <- fitness_function(new_pop,dfx,base_inicial,sel,iter)
    M <- update_umda(M,fitness_pop,attr)
  }

  return(fitness_pop)
}
```

.3.3. Método TAN

Listing 6: Método TAN

```
eda_tan_kbm2l <- function(dfx,base_inicial,iter,pop,sel) {
  #Function which uses the EDA algorithm to calculate the optimal base
  #dfx is the initial dataframe of the decision problem
  #base_inicial is the initial base
  #iter is the number of iterations of the problem
  #pop is the population to be generated
  #sel is the selection of the population after the fitness
  attr <- length(base_inicial)
  population <- generate_initial_population(pop,attr)
  for(i in 1:iter) {
    fitness_pop <- fitness_function_tan(population,dfx,base_inicial,sel)
    #categorizar la columna predictora (items kbm2l)
    #fitness_pop['items'] <- cut(fitness_pop[['items']],breaks = c
        (0,30,50,70,100,120,150,Inf),labels = c(1,2,3,4,5,6,7))
    fitness_pop['items'] <- cut(fitness_pop[['items']],breaks = c
        (0,600,750,800,850,900,950,Inf),labels = c(1,2,3,4,5,6,7))
    if(length(unique(fitness_pop$items)) == 1)
    {
      #se ha convergido a un valor
      return(fitness_pop)
    }
    population <- generate_new_pop_tan(fitness_pop,pop)
  }
  return(population)
}
```