Técnicas de extracción de información de bases de datos relacionales

Daniel Calvo Francés

Trabajo de fin de grado de Matemáticas
Universidad de Zaragoza

Director del trabajo: Jorge Lloret Gazo
enero de 2024

Resumen

Relational databases are crucial for efficiently managing information by organizing data in a structured manner and enabling relationships between them. They ensure the integrity and consistency of information. Additionally, information extraction (or data mining) techniques are essential for analyzing large datasets and gaining meaningful insights. Both tools play a crucial role in the modern world, driving informed decision-making and process optimization across various sectors, from business to scientific research. In this paperwork we will approach different data mining techniques which include: techniques for extracting classification rules, association rules and the clustering technique.

A classification rule is a logical expression that describes patterns in datasets, identifying relationships between variables and predicting the membership of an instance in a specific category or class. These rules are used in machine learning algorithms to automate the classification of data based on predefined criteria. By applying classification rules, the goal is to generalize patterns observed in training data to make accurate predictions about new instances. An algorithm that looks for classification rules is the *1-R Algorithm*, which generates a one-dimensional decision tree over an attribute and filters the rules with the lowest error.

On the other hand, association rules are logical patterns that identify relationships and correlations between variables within datasets. They reveal co-occurrences to discover hidden patterns. While association rules uncover associations and dependencies among variables, classification rules prioritize predictive accuracy for assigning instances to specific categories. Both rule types play distinct roles in data analysis, with association rules emphasizing pattern discovery and classification rules focusing on predictive modeling. These related variables that we look for are usually found in what we call frequent itemsets. These are sets of data which appear in our database with high frequency. Once we obtain the frequent itemsets of a database, the task of obtaining association rules becomes much easier. An algorithm that mines frequent itemsets is the *Apriori Algorithm*, which starts from individual items and gradually expands to larger sets, using a support threshold to filter out infrequent itemsets. There are multiple improved versions of *Apriori Algorithm*. One of such is the *FP-growth algorithm*. This algorithm is a frequent pattern mining method that efficiently discovers frequent itemsets in large datasets. It constructs a compact data structure called *FP-tree* to represent frequent patterns and exploits it to generate association rules. In contrast with *Apriori Algorithm*, *FP-growth* is particularly effective for mining frequent patterns in databases with a high volume of transactions.

Lastly, clustering is a data analysis technique that involves grouping similar data points into clusters, where items within the same cluster share common characteristics. This unsupervised learning method helps uncover patterns and structures in datasets, aiding in the exploration of inherent relationships among data points. The *K-means* algorithm is a popular clustering algorithm that partitions a dataset into K clusters by iteratively assigning data points to the cluster whose centroid is closest, then updating the centroids based on the newly formed clusters. It aims to minimize the within-cluster sum of euclidean distances.

Índice general

Capítulo 1

Nociones generales

1.1. Contexto histórico

La extracción de información de bases de datos ha experimentado una evolución significativa a lo largo de la historia, paralela al desarrollo de la tecnología de la información y la informática. Desde sus primeros pasos hasta la actualidad, esta disciplina ha sido fundamental en la gestión y análisis de datos, desempeñando un papel crucial en diversos campos, como la investigación científica, la toma de decisiones empresariales y la administración gubernamental.

En la década de 1960, con el auge de la informática, se produjo la creación de los primeros sistemas de gestión de bases de datos (SGBD). Estos sistemas establecieron las bases para la organización y recuperación eficiente de datos almacenados. Charles Bachmann contribuyó enormemente en este campo [1]. Sin embargo, la extracción de información estaba limitada principalmente a consultas básicas y operaciones de búsqueda.

Con la aparición del lenguaje de consulta estructurado (SQL) a principios de la década de 1970 y la conceptualización de bases de datos relacionales por parte de Edgar Codd [2], la extracción de información se volvió más accesible y eficaz. SQL permitió realizar consultas más complejas, estableciendo un estándar para la interacción con bases de datos relacionales, como *Oracle* y *MySQL*.

Durante los años 80 y 90, la necesidad de gestionar grandes volúmenes de datos condujo al desarrollo de almacenes de datos (*data warehousing*) [3]. Estos almacenes centralizaban datos de diversas fuentes, facilitando la extracción de información para análisis más profundos. Paralelamente, la minería de datos surgió como una disciplina que utilizaba técnicas estadísticas y de aprendizaje automático para descubrir patrones y tendencias en conjuntos de datos extensos.

La explosión de internet en la década de 1990 y el crecimiento exponencial de datos dieron paso a nuevos retos en la extracción de información. Surgieron nuevas tecnologías para abordar la gestión y análisis de grandes cantidades de datos, marcando el inicio de la era del *Big Data* [4]. La extracción de información se volvió más compleja pero también más valiosa.

En la actualidad, la extracción de información se enfrenta a desafíos como la propia diversidad de fuentes de datos, la necesidad de procesamiento en tiempo real o la protección de la privacidad. Tecnologías como la inteligencia artificial y el procesamiento del lenguaje están transformando la manera en que se extrae información, permitiendo análisis más sofisticados y personalizados.

Las bases de datos relacionales han sido una herramienta fundamental en el mundo de la gestión de la información durante décadas. Estas bases de datos estructuradas han permitido almacenar y organizar grandes cantidades de datos de manera eficiente, facilitando la gestión y manipulación de la información. Sin embargo, el verdadero valor de una base de datos radica en la información que contiene y en la capacidad de extraer conocimiento valioso de ella. Para lograr este objetivo, es necesario utilizar técnicas de extracción de información que nos permitan obtener datos significativos y relevantes de una base de datos relacional. [5] [6]

En este trabajo, exploraremos diversas técnicas utilizadas para extraer información de bases de datos relacionales. Estas técnicas nos permitirán transformar datos brutos en conocimiento útil para la toma de

decisiones, la generación de informes y el descubrimiento de información valiosa.

1.2. Tablas iniciales

Vamos a presentar, en primer lugar, algunos ejemplos de bases de datos relacionales que nos servirán para ilustrar todos los conceptos de este trabajo. En una base de datos relacional, la información se organiza en forma de tablas.

1.2.1. El tiempo atmosférico

La Tabla 1.1 representa la recogida de datos sobre si es conveniente o no jugar a determinado juego según las condiciones meteorológicas del día. Cada fila de la tabla representa un día registrado. En la columna *Pronóstico* podemos encontrar las opciones *Soleado*, *Lluvioso* o *Nublado*. Para la columna *Temperatura* el día se puede registrar como *Caluroso*, *Templado* o *Frío*. La *Humedad* puede ser *Alta* o *Normal*. En la columna *Viento* se registra la existencia de viento mediante *Verdadero* o *Falso*. Finalmente, en la columna *Jugar* registramos si la decisión tomada fue de *Sí* o *No*.

Predicción1

Pronóstico	Temperatura	Humedad	Viento	Jugar
Soleado	Caluroso	Alta	Falso	No
Soleado	Caluroso	Alta	Verdadero	No
Nublado	Caluroso	Alta	Falso	Sí
Lluvioso	Templado	Alta	Falso	Sí
Lluvioso	Frío	Normal	Falso	Sí
Lluvioso	Frío	Normal	Verdadero	No
Nublado	Frío	Normal	Verdadero	Sí
Soleado	Templado	Alta	Falso	No
Soleado	Frío	Normal	Falso	Sí
Lluvioso	Templado	Normal	Falso	Sí
Soleado	Templado	Normal	Verdadero	Sí
Nublado	Templado	Alta	Verdadero	Sí
Nublado	Caluroso	Normal	Falso	Sí
Lluvioso	Templado	Alta	Verdadero	No

Tabla 1.1: Tabla del tiempo atmosférico

Por otro lado, la Tabla 1.2 se trata de una versión alternativa de la Tabla 1.1. En ella se ha modificado la columna *Temperatura* y *Humedad* de forma que, en lugar de registrar los datos de forma nominal, se ha optado por hacerlo de forma numérica. En el caso de la temperatura, se ha registrado en grados Fahrenheit, mientras que la humedad en porcentaje. Las dos versiones del mismo problema nos servirán como ejemplos a lo largo del capítulo.

1.2.2. Las lentes de contacto

La Tabla 1.3 proporciona las condiciones bajo las que un optometrista podría optar por prescribir lentes de contacto duras, lentes de contacto blandas o no llevar lentes de contacto.

1.2.3. La cesta de la compra

La Tabla 1.4 representa una muestra de compras hechas en un supermercado con los atributos *ID, Hora, Objetos Comprados*.

Predicción2

Pronóstico	Temperatura	Humedad	Viento	Jugar
Soleado	85	85	Falso	No
Soleado	80	90	Verdadero	No
Nublado	83	86	Falso	Sí
Lluvioso	70	96	Falso	Sí
Lluvioso	68	80	Falso	Sí
Lluvioso	65	70	Verdadero	No
Nublado	64	65	Verdadero	Sí
Soleado	72	95	Falso	No
Soleado	69	70	Falso	Sí
Lluvioso	75	80	Falso	Sí
Soleado	75	70	Verdadero	Sí
Nublado	72	90	Verdadero	Sí
Nublado	81	75	Falso	Sí
Lluvioso	71	91	Verdadero	No

Tabla 1.2: Tabla del tiempo atmosférico con atributos numéricos

Lente

Edad	Prescripción de gafas	Astigmatismo	Ratio de lágrimas	Lentes recomendadas
Joven	Miope	No	Reducido	Ninguna
Joven	Miope	No	Normal	Blandas
Joven	Miope	Sí	Reducido	Ninguna
Joven	Miope	Sí	Normal	Duras
Joven	Hipermétrope	No	Reducido	Ninguna
Joven	Hipermétrope	No	Normal	Blandas
Joven	Hipermétrope	Sí	Reducido	Ninguna
Joven	Hipermétrope	Sí	Normal	Duras
Pre-presbicia	Miope	No	Reducido	Ninguna
Pre-presbicia	Miope	No	Normal	Blandas
Pre-presbicia	Miope	Sí	Reducido	Ninguna
Pre-presbicia	Miope	Sí	Normal	Duras
Pre-presbicia	Hipermétrope	No	Reducido	Ninguna
Pre-presbicia	Hipermétrope	No	Normal	Blandas
Pre-presbicia	Hipermétrope	Sí	Reducido	Ninguna
Pre-presbicia	Hipermétrope	Sí	Normal	Duras
Presbicia	Miope	No	Reducido	Ninguna
Presbicia	Miope	No	Normal	Blandas
Presbicia	Miope	Sí	Reducido	Ninguna
Presbicia	Miope	Sí	Normal	Duras
Presbicia	Hipermétrope	No	Reducido	Ninguna
Presbicia	Hipermétrope	No	Normal	Blandas
Presbicia	Hipermétrope	Sí	Reducido	Ninguna
Presbicia	Hipermétrope	Sí	Normal	Duras

Tabla 1.3: Tabla de las lentes de contacto

Transacción

ID	Hora	Objetos comprados
101	6:35	Leche, Pan, Galletas, Zumo
792	7:38	Leche, Zumo
1130	8:05	Leche, Huevos
1735	8:40	Pan, Galletas, Café

Tabla 1.4: Tabla de la cesta de la compra

1.2.4. Empleados de una empresa

La siguiente tabla almacena una muestra de empleados de una empresa con los atributos *ID*, *Edad*, *Años de servicio*.

Empleados

ID	Edad	Años de servicio
1	30	5
2	50	25
3	50	15
4	25	5
5	30	10
6	55	25

Tabla 1.5: Tabla de los empleados de la empresa

1.3. Conceptos iniciales

Antes de comenzar estudiando las diversas formas de extraer la información, debemos conocer algunas nociones elementales de bases de datos.

Definición. Una *base de datos* es una colección de datos relacionados. Generalmente, se visualiza en formato de *tablas*. Tiene las siguientes propiedades implícitas:

- Representa algún aspecto del mundo real
- Es una colección coherente de datos con significado implícito
- Se diseña con un propósito específico. Tiene un grupo predeterminado de usuarios y aplicaciones preconcebidas para esos usuarios

Las bases de datos se diseñan de acuerdo con un *modelo de bases de datos*. El modelo proporciona los materiales para crear la estructura de la base de datos. Algunos de los modelos de diseño de base de datos son:

- Nivel conceptual: modelo Entidad/Relación
- Nivel lógico: modelo relacional

En este trabajo, usaremos el modelo relacional. Vamos a definir los elementos que lo componen.

Definición. Un *esquema de relación R*, denotado por $R(A_1, A_2, ..., A_n)$, está compuesto por el nombre de la relación R y un listado de *atributos* $A_1, A_2, ..., A_n$. Llamamos *atributo* A_i a la característica o rasgo de un tipo de entidad que describe la propia entidad. Los atributos aceptan posibles *valores de atributo* que pertenecen a un conjunto de posibles valores denominado *dominio D* y se denota $dom(A_i)$.

Definición. Una *relación r* del esquema de relación $R(A_1, A_2, ..., A_n)$, denotada $r(R)$ es un conjunto de n-tuplas $r = \{t_1, t_2, ..., t_m\}$. Cada *n-tupla t* es una lista ordenada de n valores $t =< v_1, v_2, ..., v_n >$, donde v_i, con $1 \leq i \leq n$, es un valor de atributo de $dom(A_i)$ o un valor *NULO*. Un valor *NULO* representa una omisión en la entrada de datos, ya sea por ser desconocidos o porque no existen en la respectiva tupla.

Ejemplo. La Tabla 1.3 presenta el esquema de relación:

LENTE(Edad, Prescripción de gafas, Astigmatismo, Ratio de lágrimas, Lentes recomendadas)

El nombre del esquema de relación es *LENTE* y tiene cinco atributos: *Edad, Prescripción de gafas, Astigmatismo, Ratio de lágrimas, Lentes recomendadas*. Cada fila de la Tabla 1.3 se corresponde con una tupla y cada nombre de columna con un atributo. Notar que como sucede en la Tabla 1.2, los valores de atributo pueden ser numéricos.

A lo largo de este trabajo usaremos el concepto de esquema de relación y tabla de forma intercambiable.

Definición. Una *base de datos relacional* es un conjunto de esquemas de relación junto con un conjunto de relaciones, una por cada uno de los esquemas de relación.

Ejemplo. La base de datos *bdLente* está formada por el esquema de relación *Lente*. Una relación de este esquema es la formada por las veinticuatro tuplas de la Tabla 1.3

Definición. Definimos *técnica de extracción* como el proceso de encontrar información relevante a partir de unos datos específicos almacenados en una base de datos, ya sea para su análisis, informes o cualquier otro propósito. Esta técnica implica la obtención de información de una base de datos de manera organizada y estructurada.

1.4. Técnicas de extracción

Según los objetivos que tengamos al analizar una base de datos, podemos utilizar distintas técnicas de extracción. Cada una de ellas nos brindará unos patrones u otros que nos servirán para sacar conclusiones. A veces estas conclusiones nos serán utiles y otras veces no: no siempre podremos obtener la información que buscamos. Aunque a lo largo del trabajo veremos varias técnicas distintas apoyadas por ejemplos, en la práctica los problemas tienden a complicarse por múltiples factores externos e imprevistos. Muchas veces no estará claro qué técnica es beneficioso aplicar. Sin embargo, conviene conocer que tipos de técnicas de extracción podemos utilizar. En concreto, en este trabajo vamos a hablar de las reglas de clasificación, de las reglas de asociación y del *Clustering*:

- Reglas de clasificación: En las técnicas de extracción de reglas de clasificación se toma uno de los atributos de la tabla y se generan reglas respecto a los posibles valores de tal atributo. Por ejemplo, en la Tabla 1.1 podríamos buscar patrones que nos permitan averiguar si podemos jugar al juego requerido según si las condiciones de humedad son altas o normales.

- Reglas de asociación: Similares a las anteriores, la diferencia radica en que no se limitan a un solo atributo sino que se buscan reglas verosímiles para relacionar los distintos datos. Por ejemplo, en la Tabla 1.4 podríamos averiguar en base a los datos que si alguien compra leche es probable que compre zumo también.

- *Clustering*: es una técnica que busca separar los datos en grupos disjuntos donde se reúnan características similares. Por ejemplo, con los datos de la Tabla 1.5 podríamos buscar agrupar a los empleados de la empresa según su veteranía utilizando los atributos de *Edad* y *Años de servicio* para determinar posibles subidas de sueldo.

Conocidas las técnicas que vamos a tratar, vamos a centrarnos en cada una de ellas en los próximos capítulos.

Capítulo 2

Reglas de clasificación

Este capítulo está basado en el capítulo 4 del libro *DATA MINING Practical Machine Learning Tools and Techniques* de *I. H. Witten* y *E. Frank*. [7]

La primera técnica de extracción que vamos a explicar obtiene como resultado reglas de clasificación. Para entender este concepto, vamos a explicar primero ciertos conceptos clave y qué es un árbol de decisión.

Definición. Una *regla* en términos de bases de datos es un conjunto de sentencias lógicas sobre los atributos de una tabla que nos permite sacar conclusiones y tomar decisiones en torno a un atributo.

Definición. Llamamos *clase* de una regla a la colección de tuplas que contienen valores de atributo que obedecen la misma regla. La *frecuencia* de la clase es el número de tuplas del esquema de relación que pertenecen a dicha clase.

Definición. Denominamos *índice de error* a la proporción de errores cometidos por un conjunto de reglas sobre una base de datos.

Ejemplo. Veamos una regla de clasificación de la Tabla 1.1 sobre los atributos *Humedad* y *Jugar*:

$$Alta \rightarrow No$$

Esta regla declara que si la humedad es *Alta*, entonces la decisión será no jugar. Podemos observar que, en la Tabla 1.1, esto sucede en cuatro de las siete ocasiones. Por tanto, la clase de la regla $Alta \rightarrow No$, está formada por las tuplas t_1, t_2, t_8 y t_{14}.

Puesto que *Jugar* solo puede ser *Sí* o *No*, las dos clases que tenemos sobre *Humedad Alta* son $Alta \rightarrow No$ (4/7 casos) y $Alta \rightarrow Si$ (3/7 casos). Por tanto, el índice de error de la regla $Alta \rightarrow No$ es de 3/7, ya que hay tres casos de los siete donde no se cumple la regla.

Con todo esto, podemos deducir que si en nuestro pronóstico tenemos valores de humedad altos, la decisión más probable será la de no jugar al juego indicado.

Definición. Un *árbol de decisión* es un diagrama de clasificación donde se representan reglas en forma de *nodos* y *ramas*. En cada nodo del árbol se evalúa un atributo particular. Normalmente se compara con una constante, aunque en otras ocasiones se comparan dos atributos o varios mediante una función. Cada nodo está comunicado con otro mediante ramas que simbolizan los posibles resultados de la evaluación anterior. Cada nodo proporciona una clasificación (o conjunto de clasificaciones) de todas las instancias que llegan al nodo.

Cuando el nodo es un atributo nominal, el número de ramificaciones es igual al número de valores de atributo distintos. Como hay una rama por cada posible valor, el mismo atributo no volverá a aparecer a lo largo del árbol.

Cuando el atributo es numérico, la evaluación se hace normalmente por comparación de mayor, menor o igual con un valor concreto, de forma que el nodo se divide en dos ramas. A veces, por la naturaleza del problema, conviene dividir en tres ramas considerando la opción de igual por separado.

Ejemplo. Utilizando la Tabla 1.3 sobre las lentes de contacto, queremos construir un árbol de decisión que nos sirva para conocer que tipo de lentes serán recomendadas por el optometrista en base a los datos conocidos.

La Figura 2.1 representa un árbol de decisión sobre la Tabla 1.3. Podemos observar que en primer lugar se evalúa uno de los atributos de la tabla (en este caso *Ratio de lágrimas*).

Si el valor es *Reducido*, se recomendará no llevar lentes de contacto. Si el valor es *Normal*, no podemos asegurar ningún valor del atributo, así que evaluamos el atributo *Astigmatismo*.

Continuamos y vemos que si el valor de *Ratio de lágrimas* es *Normal* y el valor de *Astigmatismo* es *No*, se recomendará llevar lentes de contacto *Blandas*. Por otro lado, si el valor de *Ratio de lágrimas* es *Normal* y el valor de *Astigmatismo* es *Sí*, no podemos asegurar nada, así que pasaríamos al siguiente atributo.

Debemos observar que la construcción de cada regla se realiza desde el nodo inicial. Por ejemplo, para que el valor *No* del atributo *Astigmatismo* se evalúe en llevar lentes blandas, debemos haber obtenido antes que *Ratio de lágrimas* tenga valor *Normal*, ya que si el valor fuera *Reducido*, el resultado final es no recomendar lentes de contacto.

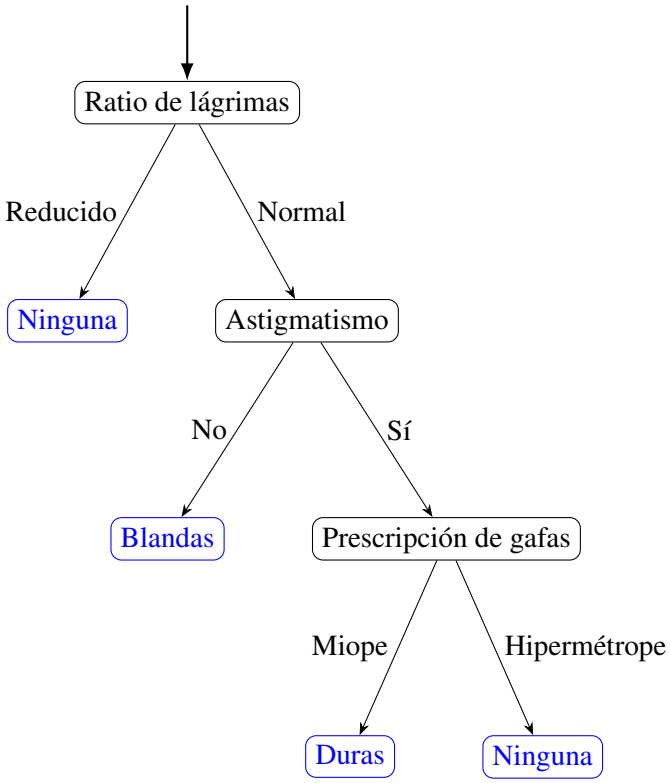


Figura 2.1: Árbol de decision para la tabla de lentes de contacto

Ahora podemos entender qué es una regla de clasificación.

Definición. Una *regla de clasificación* es un tipo de regla compuesta por un antecedente y una conclusión lógica, construída a partir de los valores de los atributos de un esquema de relación. Generalmente, los antecedentes suelen estar unidos mediante el conector lógico *Y*. Una regla de clasificación tiene la forma:

```
Si a Y b entonces x
```

Aunque puede resultar sencillo obtener una regla de clasificación de un esquema de relación simple, no resulta un problema sencillo hacerlo de una base de datos grande. Más adelante estudiaremos el algoritmo 1-R que nos servirá para obtener un conjunto fiable de reglas de clasificación de una tabla cualquiera.

Ejemplo. Es sencillo obtener un conjunto de reglas de clasificación de un árbol de clasificación. Utilizando el árbol de la Figura 2.1, podemos obtener algunas reglas como las siguientes:

```
Si Ratio de lágrimas = Reducido, entonces Lentes recomendadas = Ninguna
```

```
Si Ratio de lágrimas = Normal Y Astigmatismo = No,
entonces Lentes recomendadas = Blandas
```

2.1. Algoritmo 1-R

En muchas ocasiones, cuando tratamos de obtener información de una base de datos, resulta apropiado comenzar por lo más simple. El algoritmo 1-R (proveniente del inglés *1-rule*) es un método sencillo de obtención de reglas de clasificación de un conjunto de datos. Este algoritmo busca árboles de clasificación simples de los que se obtienen reglas de clasificación simples. Por ejemplo, para la Tabla 1.1, un árbol simple sería:

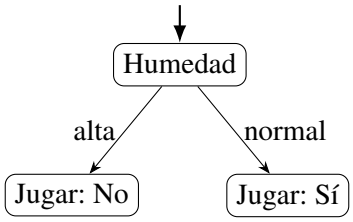


Figura 2.2: Árbol de decision simple generado mediante algoritmo 1-R

Nuestro objetivo es construir una serie de reglas de clasificación con el menor índice de error posible a partir de una regla inicial apropiada. De esta forma, obtendremos un conjunto de reglas que nos permitirán clasificar de una forma precisa la información de nuestra base de datos y así tomar decisiones pertinentes en torno a ella.

Los pasos del algoritmo son los siguientes:

```
Entrada: Atributo de clasificación fijado y resto de atributos.

   1. Generar un árbol de un nivel de decisión sobre uno de los atributos
respecto al atributo de clasificación fijado.

   2. Escoger la clase más frecuente de cada posible valor de atributo y
asignar a la clase escogida el valor de atributo correspondiente.

   3. Calcular el índice de error de esa regla.

   4. Calcular el índice de error total de las reglas resultantes del atributo
que se escogió en el primer paso.

   5. Repetir con cada atributo.

Salida: Conjunto de reglas de clasificación simples.
```

Figura 2.3: Pasos del algoritmo 1-R

Si en algún caso obtenemos un empate en el número de errores, nos quedaremos con una de las dos opciones aleatoriamente. Este criterio nos sirve para evitar problemas en la teoría, pero en la aplicación real podría ser conveniente utilizar otro criterio.

Ejemplo. Veamos la aplicación del algoritmo a la Tabla 1.1 y sigamos los pasos descritos. Queremos buscar reglas verosímiles para ver si será conveniente jugar o no utilizando los datos registrados en nuestro esquema de relación.

1. Escogemos por ejemplo el atributo *Humedad*. Puesto que en nuestra tabla los posibles valores de dicho atributo pueden ser *Alta* o *Normal*, tenemos que para cada uno, sus clases son:

Regla	**Frecuencia**
Alta → Sí	3/7
Alta → No	4/7
Normal → Sí	6/7
Normal → No	1/7

Tabla 2.1: Evaluación del atributo *Humedad* de la tabla del tiempo atmosférico

2. Nos quedamos entonces con la clase más frecuente de cada valor de atributo, es decir, con:

$$Alta \rightarrow No$$

$$Normal \rightarrow Si$$

3. Es directo ver que los índices de error de ambas reglas son 3/7 y 1/7 respectivamente, ya que las frecuencias de ambas clases son 4/7 y 6/7 respectivamente.

4. El índice de error total de la regla de clasificación del atributo *Humedad* que recoge las reglas $Alta \rightarrow No$ y $Normal \rightarrow Si$ es entonces 4/14. Una forma de verlo es plantear que de las catorce tuplas registradas, cuatro de ellas no cumplen ninguna de las dos reglas. Si hubiéramos tenido dos o más clases de un mismo valor de atributo con la misma frecuencia habríamos hecho una elección aleatoria, aunque en la aplicación real del algoritmo podría convenir otro criterio según los requerimientos del problema.

5. Aplicando el algoritmo a cada atributo, obtenemos la siguiente tabla.

	Atributo	**Reglas**	**Errores**	**Errores totales**
1	Pronóstico	Soleado → No	2/5	4/14
		Nublado → Sí	0/4	
		Lluvioso → Sí	2/5	
2	Temperatura	Caluroso → No	2/4	5/14
		Templado → Sí	2/6	
		Frío → Sí	1/4	
3	Humedad	Alta → No	3/7	4/14
		Normal → Sí	1/7	
4	Viento	Falso → Sí	2/8	5/14
		Verdadero → No	3/6	

Tabla 2.2: Evaluación de atributos de la tabla del tiempo atmosférico

En efecto podemos observar que en la regla del atributo *Temperatura: Caluroso* $\rightarrow$ *No* y en la regla del atributo *Viento: Verdadero* $\rightarrow$ *No*, se ha realizado una elección aleatoria, pues la frecuencia es también de 2/4 y de 3/6 respectivamente en las clases alternativas correspondientes.

Habiendo entonces aplicado el algoritmo 1-R a nuestro ejemplo, podemos decidir si se juega o no al deporte específico en función de las condiciones climáticas. En nuestro caso, podríamos decidir utilizando la regla del atributo *Pronóstico* o *Humedad*, ya que son las dos reglas que menos errores tienen. Si elegimos utilizando la humedad, la regla para decidir si se juega es así: si la humedad es alta, entonces no se juega.

También podemos aplicar 1-R para el caso de la existencia de nulos, en cuyo caso, simplemente tomaremos *NULO* como otro posible valor del atributo correspondiente.

2.2. Caso numérico

Adicionalmente, 1-R funciona también para el caso numérico. Podemos convertir los atributos numéricos en nominales aplicando un metodo de discretización. Veamoslo mediante un ejemplo.

Ejemplo. Utilicemos la Tabla 1.2. Vamos a ordenar los elementos del atributo *Temperatura* en una secuencia creciente con su respectivo resultado del atributo *Jugar*:

64	65	68	69	70	71	72	72	75	75	80	81	83	85
Sí	No	Sí	Sí	Sí	No	No	Sí	Sí	Sí	No	Sí	Sí	No

Para discretizar debemos particionar la secuencia. Una posible forma es colocar separaciones cada vez que cambia la secuencia.

Sí | No | Sí Sí Sí | No No | Sí Sí Sí | No | Sí Sí | No

Cada separación se corresponde con el valor medio entre el valor final e inicial de dos clases contiguas. Así pues tenemos los valores 64.5, 66.5, 70.5, 72, 77.5, 80.5 y 84. Vemos que el valor 72 nos da problemas ya que pertenece a dos clases distintas. Podemos solucionarlo moviendo la separación correspondiente a 72 una posición hacia delante y la convertimos en 73.5, obteniendo una clase mixta donde *No* es la clase mayoritaria.

Sí | No | Sí Sí Sí | No No Sí | Sí Sí | No | Sí Sí | No

Podemos entonces construir un conjunto de reglas tomando comparaciones de mayor y menor respecto a los valores de separación para decidir si el resultado será *Sí* o *No*.

Un gran problema al que nos enfrentamos es la creación de demasiadas categorías. En nuestro ejemplo vemos que solo cometiendo un error (producido por la resolución del 72 anterior) hemos creado una gran cantidad de intervalos. Una situación en la que un atributo se divida en un número excesivo de categorías, nos llevará a tener cero errores, pero esto no nos interesa. La explicación radica en que la creación de demasiadas reglas de clasificación no nos brindará ninguna información: el objetivo de las técnicas de extracción de información es encontrar reglas que engloben los distintos casos para buscar relaciones y patrones comunes para obtener información no trivial a primera vista.

Podemos entonces ser más laxos y hacer, por ejemplo, la siguiente partición:

Sí No Sí Sí Sí No No Sí Sí Sí | No Sí Sí No

En primer lugar, tenemos una partición mixta con *Sí* como clase mayoritaria y, en segundo lugar, una partición con empate de clases. Si hacemos una elección arbitraria en la segunda partición para quedarnos con *No* (en caso contrario tendríamos en total una única partición), obtenemos el siguiente conjunto de reglas con índice de error 5/14:

$$\begin{array}{lll} \texttt{Temperatura:} & \leq \texttt{77.5} & \rightarrow \texttt{Sí} \\ & > \texttt{77.5} & \rightarrow \texttt{No} \end{array}$$

De esta forma obtendríamos que, con un índice de error 5/14, si tenemos que decidir jugar o no al deporte especificado y la temperatura es menor o igual a 77.5ºF, la decisión será que sí se jugará. De la misma forma, si la temperatura supera los 77.5ºF, la decisión será no jugar (con el mismo índice de error).

Capítulo 3

Reglas de asociación

Este capítulo está basado en el capítulo 12 del libro *ADVANCES IN KNOWLEDGE DISCOVERY AND DATA MINING* de *R.Agrawal et al* [8] y en el libro *FUNDAMENTALS OF DATABASE SYSTEMS* de *Elmasri*, *Ramez* y *S. B. Navathe* [9].

Las reglas de asociación no difieren mucho de las de clasificación. La principial diferencia radica en que pueden predecir cualquier atributo, no solo la clase. Esto propicia que podamos predecir combinaciones de atributos con libertad. A diferencia de las reglas de clasificación, las reglas de asociación no buscan ser agrupadas en conjuntos de reglas sino que distintas reglas de asociación predicen distintas propiedades por sí mismas. Por tanto, si nuestro objetivo al buscar reglas de clasificación era predecir el valor de cierto atributo fijado, con las reglas de asociación nuestro objetivo será predecir relaciones consistentes entre los datos de nuestro esquema de relación.

En los apartados anteriores, hemos considerado en el modelo relacional que cada atributo es unievaluado, es decir, que acepta en cada tupla un único valor de atributo perteneciente a su dominio. Sin embargo, podemos tener atributos multievaluados, es decir, que acepten conjuntos de datos y cada posible conjunto de datos conformaría un valor de atributo diferente. El dominio de un atributo de este tipo está formado por una partición del conjunto total de datos que tengamos.

Para entender mejor lo anterior, nos vamos a apoyar en el ejemplo de la cesta de la compra. La Tabla 1.4 representa el esquema de relación *Transacción* donde se visualizan las compras hechas por clientes de un supermercado. Cada tupla del esquema es una transacción o compra independiente efectuada por un cliente. El atributo *Objetos comprados* tiene como valores conjuntos de objetos del supermercado (datos) que se han adquirido en cada transacción. Así, podríamos encontrar reglas de asociación entre los objetos del supermercado. Si observamos que *leche* y *zumo* aparecen juntos en diversas compras, podríamos suponer que existe una regla de asociación entre ambas: si alguien compra zumo, entonces comprará leche (o viceversa).

Definición. Una *regla de asociación* es una regla de la forma LI (lado izquierdo) $\Longrightarrow$ LD (lado derecho), donde $LI = \{x_1, x_2, ..., x_n\}$ e $LD = \{y_1, y_2, ..., y_m\}$ son conjuntos de datos de un mismo atributo con $LI \cap LD = \emptyset$. Llamamos *conjunto de elementos* a la unión $LI \cup LD$.

Definición. Definimos *soporte de un conjunto de datos* como la fracción de frecuencia de aparición del conjunto en nuestra muestra o base de datos. La denotamos $sup(X)$ siendo X un conjunto de datos de nuestro esquema de relación.

Definición. Definimos *soporte de una regla* $LI \Longrightarrow LD$ como la fracción de frecuencia de aparición de su conjunto de elementos correspondiente en nuestra muestra o base de datos. Es decir, es el porcentaje de tuplas de nuestro esquema de relación que contienen todos los elementos de $LI \cup LD$.

El soporte de una regla nos permite conocer lo verosimil que es dicha regla en cuanto a la cantidad de veces que los elementos de $LI \cup LD$ aparecen juntos.

Ejemplo. Usemos la Tabla 1.4 y calculemos el soporte de la regla *leche* $\Longrightarrow$ *zumo*. Es decir, queremos saber la frecuencia de aparición del respectivo conjunto de elementos {*leche, zumo*}. Como *leche* y *zumo*

aparecen juntos en dos de las cuatro transacciones, esto sigfica que ocurre el 50 % de las veces, es decir, la regla *leche* $\Longrightarrow$ *zumo* tiene soporte 0,5. Veamos ahora la regla *pan* $\Longrightarrow$ *huevos*. Puesto que *pan* y *huevos* no se han comprado juntos en ninguna de las transacciones, el soporte de la regla es 0.

Definición. Llamamos *confianza* de una regla $LI \Longrightarrow LD$ al cálculo:

$$\frac{sup(LI \cup LD)}{sup(LI)}$$

Es decir, es la probabilidad de que los elementos de *LD* aparezcan en una tupla de nuestro esquema de relación sabiendo que los elementos de *LI* aparecen. Esto nos permite medir lo fuerte o creíble que es una regla de asociación en términos de probabilidad.

Ejemplo. Veamos la Tabla 1.4 sobre la cesta de la compra comentada anteriormente. Queremos buscar alguna regla que nos permita saber si al comprar determinado artículo, se comprará otro adicional. En este ejemplo particular, vamos a simplificar considerando *LI* y *LD* como conjuntos de un único artículo.

Consideremos la regla *leche* $\Longrightarrow$ *zumo*. El respectivo conjunto de elementos $LI \cup LD$ será {*leche, zumo*}. El soporte de {*leche, zumo*} es 0,5 como hemos visto en el ejemplo anterior, mientras que el soporte de {*leche*} es 0,75. Por tanto, la confianza de *leche* $\Longrightarrow$ *zumo* es:

$$\frac{sup(\{leche, zumo\})}{sup(\{leche\})} = \frac{0,5}{0,75} = 0,67$$

Otra forma de verlo: de las tres transacciones donde aparece *leche*, en dos de ellas aparece *zumo*.

Ahora, consideramos otra regla distinta: *pan* $\Longrightarrow$ *zumo*. En este caso, $LI \cup LD$ será {*pan, zumo*}. El soporte de {*pan, zumo*} es 0,25, ya que *pan* y *zumo* aparecen juntos sólamente en una de las cuatro transacciones. Por otra parte, el soporte de {*pan*} es 0,5, pues aparece en la mitad de transacciones. Por tanto, la confianza será:

$$\frac{sup(\{pan, zumo\})}{sup(\{pan\})} = \frac{0,25}{0,5} = 0,5$$

Sólamente en una de las dos transacciones que contienen {pan} aparece *zumo*.

Si comparamos ambas reglas en términos de probabilidad, un cliente comprará zumo con una probabilidad del 66,7 % si compra leche y comprará zumo con una probabilidad del 50 % si compra pan. La primera regla tiene un soporte más alto que la segunda, por lo que existen evidencias de que la regla *leche* $\Longrightarrow$ *zumo* es más válida que la segunda regla *pan* $\Longrightarrow$ *zumo*, pues esta última ocurre tan pocas veces en el esquema de relación que no nos asegura que se vaya a cumplir o repetir.

No resulta adecuado calificar una regla como válida basándonos sólamente en lo anterior. Si obtenemos valores de soporte bajos, la regla no estará apoyada en el suficiente número de datos como para considerarla relevante. Esto será independiente de lo alta que resulte la confianza posteriormente. Podemos verlo en el siguiente ejemplo.

Ejemplo. Consideramos de nuevo la Tabla 1.4 y evaluemos la regla *huevos* $\Longrightarrow$ *leche*. Observamos que el soporte de la regla es bajo, pues tan solo ocurre en una de las cuatro transacciones, por lo que tiene soporte 0,25. Por otro lado, el soporte de {*huevos*} es de la misma forma 0,25. Calculamos la confianza:

$$\frac{sup(\{huevos, leche\})}{sup(\{huevos\})} = \frac{0,25}{0,25} = 1$$

Es decir, en el 100 % de las transacciones donde se compra *huevos* se compra *leche*. Si sólamente tuvieramos en cuenta la confianza podríamos pensar que la regla es perfecta: cualquiera que compre huevos comprará siempre leche. Pero esto no es así. Aún teniendo la máxima confianza posible, el soporte bajo nos indica que la regla se apoya en un número de casos tan mínimo que no resulta razonable aceptarla como cierta.

Teniendo en cuenta esto, necesitamos un indicador para el soporte que nos diga si la regla es válida o lo suficientemente fuerte, o si por el contrario, debemos descartarla ante la insuficiencia de casos. Con este fin, definimos:

Definición. Denominamos *umbral de aceptación* al valor de soporte mínimo para que una regla sea considerada válida. Este valor será generalmente especificado por el usuario de acuerdo a la naturaleza del problema.

Definición. Los conjuntos de elementos cuyo soporte iguala o excede el umbral de aceptación se dicen *frecuentes*.

Ejemplo. Utilizando las reglas vistas en los ejemplos previos, supongamos que el usuario ha especificado un umbral de aceptación de 0,5.

- La regla *leche* $\Longrightarrow$ *zumo* con soporte 0,5 sería una regla válida y por tanto el conjunto {*leche, zumo*} sería un conjunto frecuente de elementos.
- La regla *pan* $\Longrightarrow$ *zumo* con soporte 0,25 no se consideraría una regla válida y sería descartada.
- La regla *huevos* $\Longrightarrow$ *leche* con soporte 0,25 no se consideraría una regla válida y sería descartada.

El mayor problema que tendremos con bases de datos grandes será encontrar todos los conjuntos frecuentes de elementos con el valor de su correspondiente soporte. Para resolver esto, nos apoyaremos en algoritmos que nos faciliten su obtención. Una vez encontrados, veremos a continuación cómo tratarlos para obtener reglas de asociación sencillas.

3.1. Algoritmo Apriori

Antes de presentar el algoritmo, vamos a ver dos propiedades importantes.

Definición. Sea X conjunto frecuente de elementos. Si $\forall Y \subset X$ se cumple que Y es un conjunto frecuente de elementos, se dice que cumple la propiedad de *clausura descendente.*

Ejemplo. Consideramos la Tabla 1.4 de la cesta de la compra. Supongamos que el usuario ha especificado un umbral de aceptación de 0,5. El conjunto de elementos {*leche, zumo*} es frecuente pues tiene soporte 0,5. Igualmente, tanto el conjunto {*leche*} como el conjunto {*zumo*} tienen soporte 0,5, así que también son frecuentes y por tanto {*leche, zumo*} cumple la propiedad de clausura descendente.

Definición. Sea Y conjunto no frecuente de elementos. Si $\forall X$ tal que $Y \subset X$ se cumple que X es un conjunto no frecuente de elementos, se dice que se cumple la propiedad de *antimonotonicidad*.

Ejemplo. Consideramos de nuevo la Tabla 1.4 y tomamos un umbral de aceptación de 0,5. Podemos observar que el conjunto {*leche, huevos*} tiene soporte 0,25 y por tanto no es un conjunto frecuente. Construimos cada uno de los posibles conjuntos que contienen *leche* y *huevos*. Es sencillo observar que ninguno de estos conjuntos tendrá soporte igual o superior a 0,5 y entonces serán no frecuentes, por lo que se cumple la propiedad de antimonotonicidad.

Como comentamos previamente, uno de los mayores problemas será el encontrar los conjuntos frecuentes de una base de datos grande, ya que, si la cardinalidad es muy elevada, el cálculo del soporte de todos los conjuntos posibles es altamente costoso. Si usamos las dos propiedades anteriores en un algoritmo, el espacio combinatorio de búsqueda se reduce notablemente. Una vez tenemos los conjuntos frecuentes de elementos, resulta sencillo elaborar reglas de asociación simples a partir de los datos recogidos.

El algoritmo Apriori fue el primero en implementar las dos propiedades previas. Su funcionamiento consiste en buscar los conjuntos frecuentes de elementos. Para ello evalúa cada conjunto de menor tamaño posible para localizar los frecuentes. A continuación, aumenta el tamaño de la menor forma posible

de los conjuntos frecuentes calculados y vuelve a comprobar si son frecuentes o no. El algoritmo itera estos pasos hasta no encontrar conjuntos frecuentes. De esta forma, el algoritmo calcula una colección compuesta de todos los conjuntos frecuentes de la base de datos que satisfacen las dos propiedades anteriores y a partir de ese cálculo se elaboranr reglas de asociación respecto a los datos que aparecen en tales conjuntos. Esto permite saber qué datos tienen tendencia a aparecer juntos en la base de datos.

La entrada del algoritmo Apriori es un atributo con n datos y m tuplas. Denotaremos $L_1, L_2, ..., L_k$ a los conjuntos frecuentes de elementos. El umbral de aceptación lo denotamos u.

Los pasos del algoritmo son los siguientes:

Entrada: Atributo con n datos y m tuplas. Umbral de aceptación u.

1. Calcular el soporte de cada dato $i_1, i_2, ..., i_n$ como si se tratase de conjuntos de un solo elemento.

2. Considerar el conjunto C_1 de todos los datos $i_1, i_2, ..., i_n$ como candidato a ser el 1-conjunto frecuente de elementos.

3. Tomamos L_1 como el subconjunto de C_1 formado por los elementos i_j tales que $sup(i_j) \geq u$. Este conjunto L_1 será el 1-conjunto frecuente de elementos. Hacemos $k = 1$ e iteramos en los siguientes pasos.

4. Ahora consideramos C_{k+1} el (k+1)-conjunto frecuente de elementos candidato, formado por las combinaciones de k+1 miembros de L_k que tienen k-1 elementos en común. Adicionalmente, solo consideramos como miembros de C_{k+1} aquellos tales que cada subconjunto suyo de tamaño k aparece en L_k.

5. Formamos de nuevo L_{k+1} como el subconjunto de C_{k+1} formado por los miembros cuyo soporte supere o iguale el umbral u.

6. Si L_{k+1} está vacío, terminamos. En caso contrario, hacemos $k = k + 1$ y repetimos desde el paso 4.

Salida: Conjuntos frecuentes de elementos $L_1, L_2, ..., L_k$.

Figura 3.1: Pasos del algoritmo Apriori

Podemos observar que el algoritmo cumple la propiedad de antimonotonicidad pues si un conjunto es no frecuente el algoritmo impide que se construyan conjuntos de mayor tamaño a partir de dicho conjunto. Además, cumple la propiedad de clausura descendente ya que, por la construcción del algoritmo, si un conjunto es evaluado como frecuente significa que sus subconjuntos tuvieron que ser evaluados como frecuentes en primer lugar.

Veamos ahora un ejemplo que nos ilustre la aplicación del algoritmo.

Ejemplo. Utilizamos de nuevo la Tabla 1.4 de la cesta de la compra y consideremos el umbral de aceptación 0,5. Tenemos los datos: *leche, pan, zumo, galletas, huevos y café.*

1. Calculamos los soportes. Respectivamente son: 0,75, 0,5, 0,5, 0,5, 0,25 y 0,25.

2. El 1-conjunto candidato C_1 será {{*leche*}, {*pan*}, {*zumo*}, {*galletas*},{*huevos*}, {*café*}}.

3. Teniendo en cuenta los elementos cuyo soporte es mayor o igual que 0,5, el 1-conjunto frecuente de elementos L_1 será {{*leche*}, {*pan*}, {*zumo*}, {*galletas*}}. Vamos a crear el 2-conjunto candidato

C_2. Está formado por las combinaciones de 2 miembros de L_1 tal que no tienen elementos en común y cuyos subconjuntos aparecen todos en L_1 (ambas son triviales pues los miembros de L_1 son de un solo elemento). Tenemos pues que C_2 está formado por {*leche*, *pan*}, {*leche*, *zumo*}, {*leche*, *galletas*}, {*pan*, *zumo*}, {*pan*, *galletas*} y {*zumo*, *galletas*}.

4. Calculamos los soportes de los conjuntos anteriores. Respectivamente: 0,25, 0,5, 0,25, 0,25, 0,5 y 0,25. Por tanto, el 2-conjunto frecuente de elementos es L_2 está formado por {*leche*, *zumo*} y {*pan*, *galletas*}.

5. Puesto que L_2 no está vacío, iteramos y pasamos a buscar los 3-conjuntos frecuentes de elementos.

6. Observemos que no podemos construir C_3. Si por ejemplo tomamos {*leche*, *zumo*, *pan*}, el subconjunto {*leche*, *pan*} no está en L_2 y por tanto no podría ser un 3-conjunto frecuente de elementos por la propiedad de la clausura descendente.

Por tanto, el algoritmo termina y hemos encontrado los conjuntos frecuentes :

{leche}
{pan}
{zumo}
{galletas}
{leche, zumo}
{pan, galletas}

Podríamos entonces generar reglas de asociación utilizando los conjuntos anteriores de forma que tendrían un soporte superior al umbral de aceptación especificado. Al final del capítulo veremos cómo podemos obtener reglas de asociación de conjuntos frecuentes como los anteriores.

3.2. Árbol-PF y algoritmo de Crecimiento-PF

El mayor problema que presenta el algoritmo Apriori y derivados es que pueden generar (y por tanto tener que evaluar) un número muy elevado de conjuntos candidatos. Esto puede provocar un coste computacional muy alto en bases de datos grandes. El algoritmo Crecimiento-PF (Patrón Frecuente) es una alternativa para evitar este problema. Para ello utilizaremos un árbol de clasificación especial que denominaremos *árbol-PF*. Este tipo de árboles de clasificación sirven para almacenar la información relevante y para descubrir de manera eficiente posibles conjuntos frecuentes. Vamos a explicar su funcionamiento.

3.2.1. Construcción del árbol-PF

Para facilitar el desarrollo, vamos a considerar el soporte de los conjuntos como un conteo en vez de como una fracción.

Entrada: Base de datos

1. Calcular los soportes de todos los datos. Ordenar de forma no creciente por soporte los datos y quedarse con los frecuentes (obteniendo 1-conjuntos). Almacenarlos en una tabla T junto al contador de su soporte.

2. Para cada tupla t_i de la base de datos que contenga algunos de los 1-conjuntos frecuentes anteriores, construir otra tabla T_i donde almacenar los 1-conjuntos que aparezcan tanto en la tupla t_i como en T y guardarlos en el mismo orden no creciente en el que aparecen en T . El primer dato de esta tabla T_i será la cabecera, y el resto serán la cola.

3. Iniciar el árbol-PF con la raíz *NULL*.

4. Tomar la tabla T_i. Si del nodo actual N del árbol-PF nace una rama con un nodo cuyo nombre sea la cabecera, entonces dicho nodo pasa a ser el nodo N y se incrementa en 1 su contador asociado; en otro caso, crear un nuevo nodo N desde la raíz con un contador iniciado en 1 y ligar N con la tabla T mediante un enlace.

5. Si la cola es no vacía, la cabecera se elimina y el siguiente dato de la cola se convierte en la cabecera. Repetir el paso 4.

6. Si la cola es vacía, pasar a la siguiente tabla y repetir desde el paso 4. Si no quedan tablas T_i, terminar.

Salida: Árbol-PF y tabla T con los enlaces.

Figura 3.2: Pasos de la construcción de un Árbol-PF

Vamos a ver un ejemplo que facilite la comprensión.

Ejemplo. Utilicemos de nuevo la Tabla 1.4. Consideramos un umbral de aceptación 2 (recordamos que para esta sección usamos el soporte como un conteo en vez de una fracción). Sigamos los pasos anteriores.

1. Calculamos los 1-conjuntos frecuentes con su soporte y los ordenamos de forma no creciente: {{(*leche*, 3)}, {(*pan*, 2)}, {(*galletas*, 2)}, {(*zumo*, 2)}}. Los insertamos en la tabla T.

Objeto	**Soporte**	**Enlace**
Leche	3	
Pan	2	
Galletas	2	
Zumo	2	

Tabla 3.1: Tabla T

   De momento, la columna *enlace* aparece vacía, ya que la iremos rellenando conforme hagamos el árbol.

2. Observamos que las cuatro tuplas de la Tabla 1.4 tienen algún objeto de T, así que construimos cuatro subtablas T_i con los objetos de T en el mismo orden.

T_1	T_2	T_3	T_4
Leche	Leche	Leche	Pan
Pan	Zumo		Galletas
Galletas			
Zumo			

Tabla 3.2: Tablas T_i

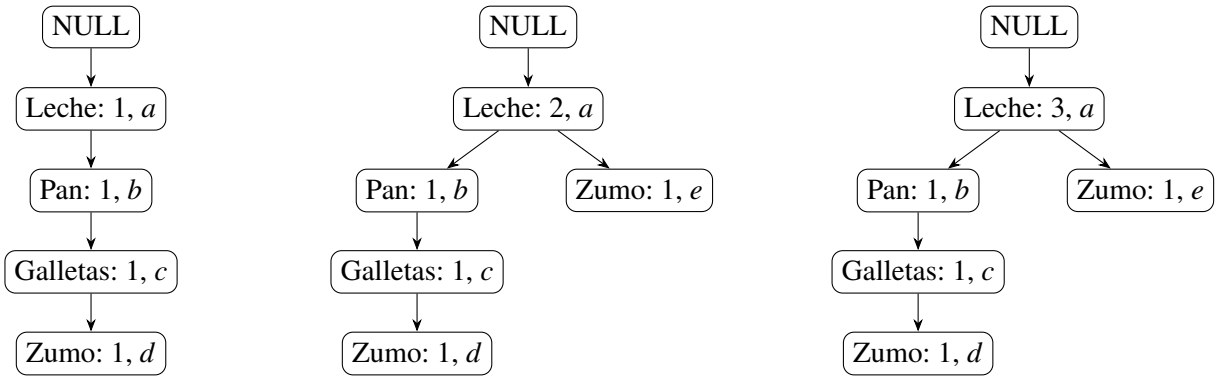
3. Iniciamos el árbol con el nodo *NULL*.

4. Tomamos T_1. Como de la raíz *NULL* no nace ninguna rama que tenga por nodo la cabecera *leche*, la creamos y le ponemos contador 1 y enlace a.

5. Eliminamos *leche* de T_1 y como la cola no está vacía, *pan* es la nueva cabecera. Repitiendo el paso anterior, no existe ninguna rama de *leche* que tenga por nodo *pan*, así que la creamos y le ponemos contador 1 y enlace b. Hacemos lo mismo con el resto de objetos de T_1.

6. Pasamos a la tabla T_2. Volvemos al nodo *NULL* y vemos que existe una rama cuyo nodo coincide con la cabecera *leche*, así que aumentamos en 1 el contador del nodo *leche* en el árbol. La nueva cabecera pasa a ser *zumo* y como no existe ninguna rama del nodo *leche* cuyo nodo sea *zumo*, creamos una nueva rama desde *leche* con el nombre *zumo* y le ponemos contador 1 y enlace e.

7. Pasamos a la tabla T_3. Volvemos al nodo *NULL* y vemos que existe una rama cuyo nodo coincide con la cabecera *leche*, así que aumentamos en 1 el contador del nodo *leche* en el árbol.



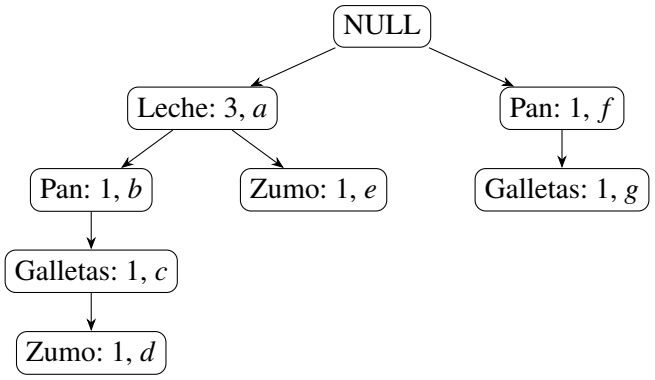
(a) T_1 (b) T_2 (c) T_3

Figura 3.3: Construcción de árbol-PF con T_1, T_2 y T_3

8. Pasamos a la tabla T_4. Volvemos al nodo *NULL* y vemos que no existe ninguna rama cuyo nodo coincida con la cabecera *pan*, así que creamos una nueva rama desde *NULL* con el nodo *pan* y le ponemos contador 1 y enlance f. De la misma forma, eliminamos *zumo* de T_4 y tomamos *galletas* como la nueva cabecera. Como no existen ramas desde el nuevo nodo anterior, creamos una nueva con el nodo *galletas* y le ponemos contador 1 y enlace g. Así, terminamos el árbol y, completando la tabla T con los enlaces de los nodos, obtenemos la Figura 3.4.

Notar que el enlace nos sirve para ubicar los contadores de cada objeto en el árbol y comprobar que no nos hemos dejado ninguno según los soportes calculados. Si la base de datos es pequeña podemos no necesitar el enlace, pero en bases de datos más grandes nos ayudará a localizar todos los datos.

Observamos, además, que el árbol-PF nos proporciona una manera más compacta de ver las transacciones relacionadas con los 1-conjuntos frecuentes.



(a) Árbol-PF de la Tabla 1.4

Objeto	Soporte	Enlace
Leche	3	*a*
Pan	2	*b*, *f*
Galletas	2	*c*, *g*
Zumo	2	*d*, *e*

(b) Tabla T con enlaces

Figura 3.4: Construcción final del Árbol-PF de la Tabla 1.4

3.2.2. Algoritmo de Crecimiento-PF

Ahora que entendemos la construcción de un árbol-PF, podemos introducir el algoritmo de Crecimiento-PF. Este algoritmo nos permite escrutar un árbol-PF dado, obtener a partir del mismo formas de relacionar los 1-conjuntos frecuentes y así poder elaborar conjuntos frecuentes de mayor tamaño.

Para poder explicar el algoritmo necesitamos dos definiciones.

Definición. Sea un nodo N de un árbol-PF al que llamaremos *sufijo*, llamaremos *base condicional* de N al conjunto de los caminos del árbol que llegan hasta N (sin incluir a N). A cada camino de este conjunto lo llamaremos *prefijo*.

Ejemplo. Tomando el árbol-PF de la Figura 3.4a, si elegimos el nodo *zumo* como sufijo, los prefijos que llegan hasta el nodo elegido son (*leche*: 1, *pan*: 1, *galletas*: 1) y (*leche: 1*), por lo que la base condicional de *zumo* sería {(*leche, pan, galletas*), (*leche*)}.

Definición. Sea un nodo N de un árbol-PF. Considerando los elementos de su base condicional como conjuntos de datos de tuplas distintas de un esquema de relación, denominamos *árbol-PF condicional* de N al árbol-PF construido a partir de la base condicional de N y con umbral de aceptación $sup(N)$.

Ejemplo. Tomamos la base condicional de *zumo* del ejemplo anterior {(*leche, pan, galletas*), (*leche*)}. Fijamos el umbral de aceptación $sup((zumo)) = 2$.

Para obtener el árbol-PF condicional de *zumo* debemos construir un árbol-PF considerando (*leche, pan, galletas*) y (*leche*) como las únicas transacciones realizadas de una tabla nueva. Como *leche* es el único objeto cuyo soporte supera el umbral 2, si seguimos los pasos de construcción de un árbol-PF podemos ver que el árbol-PF construido a partir de la base {(*leche, pan, galletas*), (*leche*)} nos brinda un único nodo (*leche*: 2). Es decir, el árbol-PF condicional de *zumo* respecto a la Tabla 1.4 es:

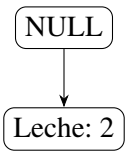


Figura 3.5: Árbol-PF condicional de *zumo* respecto a la Tabla 1.4

El algoritmo comienza evaluando si el árbol-PF introducido tiene más de una rama. Si sólamente tiene una rama, el algoritmo selecciona como conjuntos frecuentes todas las combinaciones de nodos que superan el umbral de decisión estipulado. Si el árbol-PF tiene más de una rama, comienza eligiendo el 1-conjunto frecuente menos frecuente y calcula su base condicional. A continuación calcula su árbol-PF condicional y forma los conjuntos frecuentes concatenando el sufijo con los caminos producidos en el árbol.

Los pasos del algoritmo son entonces los siguientes:

```
   Entrada: árbol-PF

   1. Analizar si el árbol-PF introducido tiene una o más ramas.

   2. Si contiene una única rama, generar los conjuntos frecuentes a partir
de todas las combinaciones de nodos que superan el umbral de aceptación
introducido.

   3. Si contiene dos o más ramas, para cada nodo en orden no decreciente de
soporte fijar su valor de soporte y hacer:

       a) Calcular su base condicional.

       b) Calcular su árbol-PF condicional.

       c) Si dicho árbol-PF es no vacío, generar los conjuntos frecuentes
       concatenando sus caminos con el nodo inicial. Calcular su soporte.

Salida: Conjuntos frecuentes con su soporte.
```

Figura 3.6: Pasos del algoritmo de Crecimiento-PF

Veamos pues el funcionamiento del algoritmo con un ejemplo:

Ejemplo. Tomamos el árbol de la Figura 3.4a y determinamos umbral de aceptación 2.

1. El árbol-PF tiene más de una rama.
2. Como tiene más de una rama ignoramos el paso 2.
3. Comenzamos con el objeto *zumo* y fijamos su valor de soporte, que es 2.
   - *a*) Construimos su base condicional. Sabemos por los ejemplos anteriores que es {(*leche, pan, galletas*), (*leche*)}.
   - *b*) Construimos su árbol-PF condicional. De igual manera, sabemos por el ejemplo anterior que está compuesto de un único nodo (*leche: 2*).
   - *c*) Como el árbol-PF condicional es no vacío y tiene un único nodo, el único conjunto frecuente generado es {*leche, zumo*} con soporte 2.

   Pasamos al objeto *galletas* y fijamos su soporte 2.

   - *a*) Construimos su base condicional. Los prefijos que llegan hasta *galletas* son (*leche*: 1, *pan*: 1) y (*pan*: 1), por lo que la base condicional de *galletas* será {(*leche, pan*), (*pan*)}.
   - *b*) Construimos su árbol-PF condicional. Es sencillo observar que está compuesto de un único nodo: (*pan*: 2).
   - *c*) Como el árbol-PF condicional es no vacío y tiene un único nodo, el único conjunto frecuente generado es {*pan, galletas*} con soporte 2.

   Si vamos ahora con los objetos *pan* y *leche* es sencillo ver que sus respectivos árboles-PF condicionales quedan vacíos, así que no generan conjuntos frecuentes. Así, el algoritmo finaliza y nos devuelve los siguientes conjuntos frecuentes con su soporte: {*leche*: 3}, {*pan*: 2}, {*galletas*: 2}, {*zumo*: 2}, {*leche, zumo*: 2} y {*pan, galletas*: 2}. Con estos conjuntos podremos elaborar fácilmente reglas de asociación verosímiles para nuestra base de datos.

3.3. Obtención de reglas de asociación

Los algoritmos anteriores nos permiten averiguar los conjuntos frecuentes de una base de datos dada. Necesitamos saber cómo tratar estos conjuntos para obtener de ellos reglas de asociación. Para ello, vamos a ver primero el siguiente resultado.

Proposición. *Sea X un conjunto frecuente de elementos y sea $Y \subset X$. Sea $Z = X \setminus Y$. Entonces, si $sup(X)/sup(Z) > c$ con c una confianza mínima establecida, la regla $Z \Longrightarrow Y$ es una regla válida.*

Demostración. Sea c una confianza mínima establecida. Sea $X = \{x_1, ..., x_n\}$ un conjunto frecuente de elementos. Sea $Y = \{x_1, ..., x_m\}$ con $m < n$, cambiando el orden de los elementos si es necesario. Entonces, $Z = \{x_{m+1}, ..., x_n\}$ y por tanto es trivial que $X = Z \cup Y$.

Asumiendo que $sup(X)/sup(Z) > c$, esto es equivalente a que $sup(Z \cup Y)/sup(Z) > c$. Por tanto, la confianza de la regla $Z \Longrightarrow Y$ supera la confianza mínima establecida y entonces es una regla válida. □

Utilizando la proposición anterior, veamos un sencillo algoritmo que nos genera reglas de asociación:

```
Entrada: Conjunto frecuente X y confianza mínima establecida c

   1. Generar todos los subconjuntos no vacíos Y_i de X

   2. Para cada i, construímos Z_i = X \ Y_i y calculamos c_i = sup(X)/sup(Z_i).

   3. Para cada j tal que c_j > c, creamos la regla de asociación Z_j ⟹ Y_j

Salida: Reglas de asociación.
```

Figura 3.7: Pasos del algoritmo de obtención de reglas de asociación de conjuntos frecuentes

Ejemplo. Partimos del conjunto frecuente $\{leche,\ zumo\}$ obtenido con el algoritmo de Crecimiento-PF en el ejemplo anterior. Consideramos $c = 0{,}6$.

1. Los subconjuntos no vacíos son $Y_1 = \{leche\}$ y $Y_2 = \{zumo\}$.
2. Tomando por ejemplo Y_1, tenemos que $Z_1 = \{leche,\ zumo\} \setminus \{leche\} = \{zumo\}$, luego

$$c_1 = \frac{sup(\{leche,\ zumo\})}{sup(\{zumo\})} = \frac{0{,}5}{0{,}5} = 1$$

3. Como $c_1 > c$, tenemos que la regla $Z_1 \Longrightarrow Y_1$ es decir $\{zumo\} \Longrightarrow \{leche\}$ es una regla de asociación válida.

Habiendo obtenido esta regla, podemos suponer que si un cliente compra zumo, entonces es muy posible que compre leche.

Capítulo 4

Clustering

Este capítulo está basado en el libro *DATA MINING Practical Machine Learning Tools and Techniques* de *I. H. Witten* y *E. Frank* [7] y en el libro *FUNDAMENTALS OF DATABASE SYSTEMS* de *Elmasri*, *Ramez* y *S. B. Navathe* [9].

En las reglas de clasificación vistas anteriormente sucedía lo que conocemos como *aprendizaje supervisado*, es decir, dada una base de datos, los resultados de nuestras reglas de clasificación se daban de acuerdo a una clasificación estipulada previamente. Por ejemplo, en la Tabla 1.3 de las lentes de contacto se podía recomendar una de tres opciones: no llevar lentes, llevar duras o llevar blandas. El *Clustering* (traducido como agrupamiento) es una técnica de extracción de información basada en el *aprendizaje no supervisado* (es decir, los datos de nuestra muestra no se reparten en una clasificación previa) cuyo objetivo es clasificar los datos en grupos disjuntos, de tal forma que los datos recogidos en un mismo grupo son similares y los datos pertenecientes a grupos distintos son poco semejantes.

La pregunta natural que nos surge es cómo podemos diferenciar los datos de la muestra para clasificarlos en grupos. La respuesta a esto no es sencilla pues existen muchas formas distintas de *Clustering*. Algunas de ellas son: según una jerarquía especificada por el usuario; según la probabilidad que tienen de pertenecer a un grupo, utilizando teoría de grafos analizando cómo están conectados los datos entre sí y clasificándolos por número de conexiones, o usando teoría espectral representando los datos en matrices de similitud.

Nosotros vamos a centrarnos en el método de *Clustering* por distancia en datos numéricos. Para ello debemos definir primero qué es una distancia.

Definición. Sea X un conjunto de elementos, llamamos *distancia* a la aplicación $d\colon X \times X \to \mathbb{R}$ que satisface:

- $d(a,b) \geq 0, \quad \forall a,b \in X$
- $d(a,b) = 0 \iff a = b, \quad \forall a,b \in X$
- $d(a,b) = d(b,a), \quad \forall a,b \in X$
- $d(a,c) \leq d(a,b) + d(b,c), \quad \forall a,b,c \in X$

En particular, usaremos la distancia euclidiana:

Definición. Sea X un conjunto de elementos n-dimensional. Sean $r_j, r_k \in X$. La distancia euclidiana entre dos puntos se calcula:

$$d(r_j, r_k) = \sqrt{|r_{j1} - r_{k1}|^2 + |r_{j2} - r_{k2}|^2 + \ldots + |r_{jn} - r_{kn}|^2}$$

Cuanto menor sea la distancia euclidiana entre dos datos, mayor será la similaridad entre los dos. A continuación vamos a presentar un algoritmo de *Clustering* clásico que utiliza la distancia euclidiana.

4.1. Algoritmo k-medias

Disponemos de una base de datos con n tuplas y queremos tener en consideración m atributos para la aplicación del *clustering*. El algoritmo k-medias tiene como objetivo agrupar esas tuplas en k *clusters* (o grupos). Para ello, consideraremos las tuplas como datos de dimensión m. Así, el algoritmo elige aleatoriamente k datos que serán los centroides (medias). Cada dato se clasifica en un *cluster* calculando la distancia euclidiana y, a continuación, se recalculan los centroides de cada grupo. Este proceso se itera, de forma que cada iteración es más óptima que la anterior.

Para decidir si una solución es mejor que otra, definimos una medida de error:

Definición. Sea una base de datos con datos $r_1,...,r_n$ de dimensión m. Si queremos clasificar en k *clusters* $C_1,...,C_k$, cada uno con media $m_1,...,m_k$ respectivamente, definimos el error E como el calculo:

$$E = \sum_{i=1}^{k} \sum_{\forall r_j \in C_i} d(r_j, m_i)^2$$

Como hemos comentado, cada iteración es más óptima que la anterior (es decir, se va reduciendo el error) de forma que el algoritmo siempre converge. Elegidos el número deseado de *clusters* k y sabiendo el número n de datos junto a su dimensión m (recordar que m será el número de atributos considerados), veamos los dos pasos del algoritmo.

Entrada: Número de *clusters* k deseado y atributos en consideración.

1. Elegir aleatoriamente k datos que serán los centroides $m_1,...,m_k$ de los *clusters* $C_1,...,C_k$

2. Iteramos los siguientes pasos hasta la convergencia (es decir, hasta que no haya cambios en la asignación de datos a *clusters*):

   a) Calcular la distancia euclidiana de cada dato r_i a cada centroide.

   b) Asignar cada dato r_i al *cluster* C_j donde la distancia de r_i a m_j es la mínima.

   c) Recalcular el centroide de cada *cluster* con el siguiente cálculo, donde n es el número de datos y m el número de dimensiones. Así, el centroide del *cluster* C_i queda:

$$\bar{m_i} = \left(\frac{1}{n} \sum_{\forall r_j \in C_i} r_{j1}, ..., \frac{1}{n} \sum_{\forall r_j \in C_i} r_{jm} \right)$$

Salida: k *Clusters*

Figura 4.1: Pseudocódigo para el algoritmo k-Medias

Ejemplo. Tomamos la Tabla 1.5 de los empleados de una empresa. En la tabla aparecen el identificador, la edad y los años de servicio de una serie de empleados. Nuestro objetivo es buscar una clasificación de veteranía de los empleados, de forma que se tenga en cuenta tanto la edad del empleado como sus años de servicio en la empresa.

Vamos a aplicar el algoritmo a los atributos *Edad* y *Años de servicio* y vamos a buscar repartir los empleados en dos *clusters* ($k = 2$).

1. Asumamos que el algoritmo ha elegido aleatoriamente los empleados *ID* 3 (50, 15) e *ID* 6 (55, 25) para ser los centroides m_1 y m_2 de los *clusters* C_1 y C_2 respectivamente.

2. Veamos la primera iteración:
   a) La distancia de *ID* 1 a m_1 es 22,4 y a m_2 es 32. La distancia de *ID* 2 a m_1 es 10 y a m_2 es 5. La distancia de *ID* 4 a m_1 es 25,5 y a m_2 es 36,6. La distancia de *ID* 5 a m_1 es 20,6 y a m_2 es 29,2.
   b) Por tanto el *cluster* C_1 es {*ID* 1, *ID* 3, *ID* 4, *ID* 5} y C_2 es {*ID* 2, *ID* 6}
   c) Recalculamos los centroides con la fórmula anterior y tenemos que $\bar{m_1} = (33{,}75; 8{,}75)$ y $\bar{m_2} = (52{,}5; 25)$.

   Si calculamos el error de esta primera operación, nos saldría:

$$E = d(ID1, \bar{m_1})^2 + d(ID3, \bar{m_1})^2 + d(ID4, \bar{m_1})^2 + d(ID5, \bar{m_1})^2 + d(ID2, \bar{m_2})^2 + d(ID6, \bar{m_2})^2 = 449{,}93$$

3. Hacemos la segunda iteración:
   a) La distancia de *ID* 1 a $\bar{m_1}$ es 5,3 y a $\bar{m_2}$ es 30,1. La distancia de *ID* 2 a $\bar{m_1}$ es 22,98 y a $\bar{m_2}$ es 2,5. La distancia de *ID* 3 a $\bar{m_1}$ es 17,41 y a $\bar{m_2}$ es 10,3. La distancia de *ID* 4 a $\bar{m_1}$ es 9,52 y a $\bar{m_2}$ es 34. La distancia de *ID* 5 a $\bar{m_1}$ es 3,95 y a $\bar{m_2}$ es 27,04. La distancia de *ID* 6 a $\bar{m_1}$ es 26,75 y a $\bar{m_2}$ es 2,5.
   b) Por tanto el *cluster* C_1 ahora es {*ID* 1, *ID* 4, *ID* 5} y C_2 es {*ID* 2, *ID* 3, *ID* 6}.
   c) Recalculamos los centroides y son $\bar{\bar{m_1}} = (28{,}3; 6{,}7)$ y $\bar{\bar{m_2}} = (51{,}7; 21{,}7)$.

   Si calculamos el error de esta segunda operación, obtenemos:

$$E = d(ID1, \bar{\bar{m_1}})^2 + d(ID4, \bar{\bar{m_1}})^2 + d(ID5, \bar{\bar{m_1}})^2 + d(ID2, \bar{\bar{m_2}})^2 + d(ID3, \bar{\bar{m_2}})^2 + d(ID6, \bar{\bar{m_2}})^2 = 116{,}25$$

4. Si hacemos otra iteración, comprobaremos que todos los datos se mantienen en los *clusters* anteriores. Por tanto, terminamos el algoritmo.

Así, hemos obtenido los *clusters* C_1 = {*ID* 1, *ID* 4, *ID* 5} y C_2 = {*ID* 2, *ID* 3, *ID* 6}. El primer *cluster* se corresponde con los empleados menos veteranos, habiendo tenido en cuenta su edad y sus años trabajando en la empresa. El segundo *cluster* se correspondería con los trabajadores más veteranos.

Con la aplicación del algoritmo k-Medias hemos conseguido una clasificación precisa en dos grupos con los parámetros anteriores reduciendo ampliamente el error.

Aunque el algoritmo minimiza con efectividad los mínimos locales de datos a centroides, no hay garantía de alcanzar un mínimo global. En consecuencia, el mayor problema que nos presenta el algoritmo es la alta sensibilidad que tiene a la elección de los centroides iniciales: en muestras más grandes pueden resultar *clusters* totalmente diferentes según la elección inicial de medias. Una solución rudimentaria y frecuentemente usada es repetir el algoritmo varias veces variando los centroides iniciales y escoger el resultado con el mejor mínimo global.

Bibliografía

[1] C.W.BACHMANN, *THE PROGRAMMER AS NAVIGATOR*. 16(11), 635-658, Communications of the ACM, 1973

[2] E.F. CODD, *Relational database: a practical foundation for productivity*, 25(2), 109–117, Communications of the ACM, 1982

[3] B. DEVLIN, *Thirty Years of Data Warehousing*, 23 (1), BUSINESS INTELLIGENCE, 2018

[4] J.C. MASHEY, *Big Data and the Next Wave of InfraStress*, 1998

[5] C.J.DATE, *THE DATABASE RELATIONAL MODEL: A Retrospective Review and Analysis*, 1.ª ed., Prentice Hall, 2000

[6] S. SARAWAGI, *INFORMATION EXTRACTION. Foundations and Trends® in Databases*, 1(3), 261-377, 2008

[7] I. H. WITTEN Y E. FRANK, *DATA MINING Practical Machine Learning Tools and Techniques*, 2.ª ed., Elsevier, 2005

[8] R. AGRAWAL, H. MANNILA, R. SRIKANT, H. TOIVONEN, A. I. VERKAMO, *ADVANCES IN KNOWLEDGE DISCOVERY AND DATA MINING*, *Fast discovery of association rules*, *12*(1), 307-328, AAAI/MIT Press Menlo Park, CA, 1996.

[9] ELMASRI, RAMEZ AND S. B. NAVATHE, *FUNDAMENTALS OF DATABASE SYSTEMS*, Addison-Wesley, 2011

Apéndice. Implementación en PL/SQL.

A.1 Tipos utilizados

```
create or replace TYPE namesType AS VARRAY(50) OF VARCHAR2(100);
create or replace TYPE centroidArrayType IS TABLE OF NUMBER;
```

A.2 Algoritmo 1-R

```
create or replace PROCEDURE findRules1R(tabla IN varchar2, clase IN varchar2)
IS
    nombresColumna namesType :=namesType();
    TYPE cTipoCursor IS REF CURSOR;
     c_cursor cTipoCursor;
     v_column VARCHAR2(100);
     v_clase VARCHAR2(100);
    sqlString VARCHAR2(200);
     v_esta2 BOOLEAN;
    max_total NUMBER;
    max_attribute CUENTA.ATTRIBUTEVALUE%TYPE;
BEGIN
    nombresColumna:=findColumns(tabla);
    FOR i IN 1..nombresColumna.count LOOP
        IF(nombresColumna(i)<>clase) THEN
            sqlString:= 'SELECT ' || nombresColumna(i) || ', ' || clase || ' FROM ' || tabla;
            OPEN c_cursor FOR sqlString;
            LOOP
                FETCH c_cursor INTO v_column, v_clase;
                EXIT WHEN c_cursor
/* Devuelve true si hay alguna fila en la tabla cuenta que en las columnas attributevalue y clasevalue
toma los valores v_column y v_clase respectivamente. Devuelve false en otro caso*/
                v_esta2:=BELONGSTO2COLUMNsOFTABLE(v_column, v_clase, 'cuenta',
'attributeValue', 'clasevalue');
                IF (v_esta2) THEN
                    UPDATE cuenta
                    SET total=total+1
                    WHERE attributevalue=v_column AND clasevalue=v_clase;
                ELSE
                    INSERT INTO cuenta VALUES(v_column, v_clase, 1, nombresColumna(i));
                END IF;
            END LOOP;
            CLOSE c_cursor;
        END IF;
```

```
        END LOOP;
    –showRules1R:
        FOR r IN (
            SELECT ATTRIBUTEVALUE, MAX_TOTAL
            FROM(
                SELECT ATTRIBUTEVALUE, ATTRIBUTE, MAX(TOTAL) AS MAX_TOTAL
                FROM CUENTA
                GROUP BY ATTRIBUTEVALUE, ATTRIBUTE
            )
            WHERE ATTRIBUTE IN (
                SELECT ATTRIBUTE
                FROM
                    (SELECT ATTRIBUTE, SUM(MAX_TOTAL2) AS SUMA
                    FROM
                        (SELECT ATTRIBUTEVALUE, ATTRIBUTE, MAX(TOTAL) AS MAX_TOTAL2
                        FROM CUENTA
                        GROUP BY ATTRIBUTEVALUE, ATTRIBUTE)
                    GROUP BY ATTRIBUTE)
                WHERE SUMA = (SELECT MAX(SUMA)
                    FROM
                    (SELECT ATTRIBUTE, SUM(MAX_TOTAL2) AS SUMA
                     FROM
                        (SELECT ATTRIBUTEVALUE, ATTRIBUTE, MAX(TOTAL) AS MAX_TOTAL2
                        FROM CUENTA
                        GROUP BY ATTRIBUTEVALUE, ATTRIBUTE)
                    GROUP BY ATTRIBUTE
                    ))))
        LOOP
    – Almacenar el valor máximo y el ATTRIBUTEVALUE correspondiente
            max_total := r.MAX_TOTAL;
            max_attribute := r.ATTRIBUTEVALUE;
    – Mostrar las filas que cumplen con las condiciones
            FOR row_data IN (SELECT *
                FROM CUENTA
                WHERE ATTRIBUTEVALUE = max_attribute AND TOTAL = max_total AND ROW-
NUM = 1)
            LOOP
                DBMS_OUTPUT.PUT_LINE('Si ('|| row_data.ATTRIBUTE || ' = ' || row_data.ATTRIBUTEVALUE
|| ') ENTONCES ' || clase || ' = ' ||row_data.CLASEVALUE);
            END LOOP;
        END LOOP;
        DELETE FROM CUENTA;
        COMMIT;
    END;
```

A.2.1 Subalgoritmos utilizados en Algoritmo 1-R

A.2.1.1 FindColumns

```
    create or replace FUNCTION findColumns(tabla IN varchar2)
    RETURN namesType IS
        var_atributos namesType:=namesType();
```

```
BEGIN     FOR col IN (SELECT column_name FROM user_tab_columns WHERE table_name =
tabla)
    LOOP
        var_atributos.EXTEND;
        var_atributos(var_atributos.LAST):= col.column_name;
    END LOOP;
    RETURN var_atributos;
END findColumns;
```

A.2.1.2 BELONGSTO2COLUMNsOFTABLE

```
create or replace FUNCTION BELONGSTO2COLUMNsOFTABLE(v_column in varchar2, v_clase
in varchar2, tabla in varchar2, atributo in varchar2, clase in varchar2)
RETURN boolean
IS
    v_count NUMBER;
BEGIN
– Utilizamos una consulta COUNT para contar las filas que cumplen con la condición
    EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM ' || tabla ||
        ' WHERE ' || atributo || ' = :1 AND ' || clase || ' = :2'
        INTO v_count
        USING v_column, v_clase;
– Si el conteo es mayor que cero, devuelve TRUE, de lo contrario, devuelve FALSE
    RETURN v_count >0;
END BELONGSTO2COLUMNsOFTABLE;
```

A.2.2 Tablas auxiliares para Algoritmo 1-R

```
create TABLE CUENTA (ATTRIBUTEVALUE in VARCHAR2(200), CLASEVALUE in VARCHAR2(200),
TOTAL in NUMBER, ATTRIBUTE in VARCHAR2(100))
```

A.3 Algoritmo Apriori

```
create or replace PROCEDURE findSetsApriori(tabla in varchar2, atributo in varchar2, id_atributo
in varchar2, umbral in number)
IS
    sqlString1 VARCHAR2(300);
    sqlString2 VARCHAR2(300);
    TYPE cTipoCursor IS REF CURSOR;
    c_cursor1 cTipoCursor;
    c_cursor2 cTipoCursor;
    var_item varchar2(30);
    var_sop number;
    var_row namesType;
    v_repetido namestype;
    contador number:=0;
    sopor_temp number:=0;
    v_esta2 BOOLEAN;
    v_esta2_2 BOOLEAN;
    v_esta2_1 BOOLEAN;
    salir BOOLEAN;
    idposicion number;
```

```
BEGIN
– BUSCAMOS LOS 1-CONJUNTOS FRECUENTES y los añadimos a la tabla ITEMS
   sqlString1:= 'SELECT item, count(item) as frec
      FROM(SELECT ' ||id_atributo||', COLUMN_VALUE AS item
      FROM ' ||tabla||', TABLE('||tabla||'.'||atributo||'))
      GROUP BY item';
   OPEN c_cursor1 FOR sqlString1;
   LOOP
      FETCH c_cursor1 INTO var_item, var_sop;
      EXIT WHEN c_cursor1
      IF (var_sop >= umbral) THEN
         INSERT INTO ITEMS VALUES (1, namestype(var_item), var_sop);
      END IF;
   END LOOP;
   CLOSE c_cursor1;
–Iteramos k=2,... para buscar los k-conjuntos frecuentes
   FOR k in 2..100 LOOP
–Buscamos extender los varrays de tamaño k-1 y añadirles items individuales que estén en la tabla
ITEMS
      FOR fila in (SELECT CONJUNTO FROM ITEMS WHERE ITERACION=k-1) LOOP
         var_row:=fila.CONJUNTO;
         var_row.EXTEND;
– Añadimos cada item i al conjunto que queremos extender para comprobar si es frecuente o no
            FOR i in (SELECT CONJUNTO FROM ITEMS WHERE ITERACION=1) LOOP
               var_row(var_row.LAST):=i.CONJUNTO(1);
/* Para cada fila de la compra donde pertenezca el primer item del conjunto vamos a comprobar si la
extension pertenece tambien a la fila */
               sqlString2:='SELECT '||id_atributo||' FROM '||tabla||' WHERE '''||var_row(1)||''' in
(SELECT COLUMN_VALUE FROM TABLE('||tabla||'.'||atributo||'))';
               OPEN c_cursor2 FOR sqlString2;
               LOOP
                  FETCH c_cursor2 INTO idposicion;
                  EXIT WHEN c_cursor2%NOTFOUND;
                  FOR j in 2..var_row.count LOOP
                     FOR n in 1..j LOOP
– Evaluamos si el conjunto ya existe en la tabla ITEMS (sin importar el orden)
                        FOR repetido in (SELECT CONJUNTO FROM ITEMS) LOOP
                           v_repetido:=repetido.CONJUNTO;
                           v_esta2:=son_conjuntos_iguales(var_row, v_repetido);
– Si ya existe pasamos al siguiente conjunto
                           IF (v_esta2) THEN
                              salir:=TRUE;
                              EXIT;
                           END IF;
                           END LOOP;
                        IF (salir) THEN
                           EXIT;
                        END IF;
–Si llegamos aqui es que el conjunto no existe aun en ITEMS
                        v_esta2_1:=NOTBELONGS2LIST(var_row(n), tabla , idposicion, tabla ||'.'||
atributo , id_atributo);
```

```
                    v_esta2_2:=hay_valores_repetidos(var_row);
/* Si el conjunto tiene valores repetidos o el elemento n del conjunto no esta en la misma fila comprobada que el elemento 1, se sale del bucle*/
                    IF (v_esta2_1) THEN
                        EXIT;
                    ELSIF (v_esta2_2) THEN
                        EXIT;
                    ELSE
                        contador:=contador+1;
                    END IF;
                END LOOP;
/* Si todos los items del conjunto pertenecen a una misma fila de nuestra tabla, el soporte aumenta en 1*/
                IF (contador = var_row.count) THEN
                    sopor_temp:=sopor_temp+1;
                END IF;
                contador:=0;
            END LOOP;
        END LOOP;
        CLOSE c_cursor2;
/* Cuando acaba la evaluacion, si el soporte es mayor o igual al umbral, se añade el conjunto a la tabla ITEMS*/
        IF (sopor_temp >= umbral) THEN
            INSERT INTO ITEMS values (k, var_row, sopor_temp);
        END IF;
        sopor_temp:=0;
    END LOOP;
  END LOOP;
END LOOP;
–Mostramos los resultados
  DBMS_OUTPUT.PUT_LINE('LOS CONJUNTOS FRECUENTES CON SU SOPORTE SON:');
  FOR conj_frec in (SELECT CONJUNTO, SOPORTE FROM ITEMS) LOOP
    DBMS_OUTPUT.PUT_LINE('————————————————————————————');
    DBMS_OUTPUT.PUT_LINE('Conjunto:');
    FOR m in 1..conj_frec.CONJUNTO.count LOOP
      DBMS_OUTPUT.PUT_LINE(' ' || conj_frec.CONJUNTO(m));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Soporte:' || conj_frec.SOPORTE );
  END LOOP;
  DELETE FROM ITEMS;
  COMMIT;
END findSetsApriori;
```

A.3.1 Subalgoritmos utilizados en Algoritmo Apriori

A.3.1.1 son_conjuntos_iguales

```
create or replace FUNCTION son_conjuntos_iguales (p_varray1 IN namestype, p_varray2 IN namestype) RETURN BOOLEAN
  IS
    v_match_count INTEGER := 0;
  BEGIN
```

```
– Verificar si la cantidad de elementos es la misma
    IF p_varray1.COUNT <>p_varray2.COUNT THEN
        RETURN FALSE;
    END IF;
– Verificar si todos los elementos de p_varray1 están en p_varray2
    FOR i IN 1..p_varray1.COUNT LOOP
        FOR j IN 1..p_varray2.COUNT LOOP
            IF p_varray1(i) = p_varray2(j) THEN
                v_match_count := v_match_count + 1;
                EXIT;
            END IF;
        END LOOP;
    END LOOP;
– Si el número de coincidencias es igual a la cantidad total, son iguales
    RETURN v_match_count = p_varray1.COUNT;
END son_conjuntos_iguales;
```

A.3.1.2 NOTBELONGS2LIST

```
create or replace FUNCTION NOTBELONGS2LIST (v_char in varchar2, v_tabla in varchar2, v_id
in number, v_namesType in varchar2, v_id_tabla in varchar2) RETURN BOOLEAN
    IS
        v_count NUMBER;
    BEGIN
        EXECUTE IMMEDIATE 'SELECT COUNT(*) FROM (SELECT ' ||v_id_tabla || ', COLUMN_VALUE
FROM ' || v_tabla || ', TABLE('|| v_namesType ||')) WHERE ' || v_id_tabla || ' = '||v_id||' AND '''|| v_char
||''' NOT IN (SELECT COLUMN_VALUE FROM ' || v_tabla || ', TABLE('|| v_namesType ||')WHERE
' || v_id_tabla || '='||v_id||')'
        INTO v_count;
        RETURN v_count>0;
    END NOTBELONGS2LIST;
```

A.2.1.3 hay_valores_repetidos

```
create or replace FUNCTION hay_valores_repetidos(p_varray IN namesType) RETURN BOOLEAN
    IS
        l_count NUMBER;
    BEGIN
– Contar los valores distintos en el varray
    SELECT COUNT(DISTINCT column_value) INTO l_count
    FROM TABLE(p_varray);
– Compara el conteo con la longitud del varray
    RETURN l_count <>p_varray.COUNT;
    END hay_valores_repetidos;
```

A.3.2 Tablas auxiliares para Algoritmo Apriori

```
create TABLE ITEMS (ITERACION in NUMBER, CONJUNTO in NAMESTYPE, SOPORTE in
NUMBER)
```

A.4 Algoritmo K-medias para dimensión 2

```
create or replace PROCEDURE k_means(tabla VARCHAR2, k NUMBER, column1 VARCHAR2,
column2 VARCHAR2)
    IS
        v_centroid centroidArrayType:=centroidArrayType(NULL, NULL);
        v_centroid2 centroidArrayType:=centroidArrayType(NULL, NULL);
        v_fila centroidArrayType:=centroidArrayType(NULL, NULL, NULL);
        v_punto centroidArrayType:=centroidArrayType(NULL, NULL);
        v_changed BOOLEAN := TRUE;
        v_cluster NUMBER;
        v_cluster_nuevo NUMBER;
        v_id number;
        v_sql VARCHAR2(4000);
        v_sql2 VARCHAR2(4000);
        v_data_cursor SYS_REFCURSOR;
        v_data_cursor2 SYS_REFCURSOR;
    BEGIN
    – Preparamos la tabla CLUSTERS
        v_sql := 'SELECT id FROM '|| tabla;
        OPEN v_data_cursor FOR v_sql;
        LOOP
            FETCH v_data_cursor INTO v_id;
            EXIT WHEN v_data_cursor
            INSERT INTO CLUSTERS VALUES (v_id, 0);
        END LOOP;
        CLOSE v_data_cursor;
    – Construir la consulta dinámica
        v_sql := 'SELECT '||column1||', '||column2||' FROM ' || tabla;
        OPEN v_data_cursor FOR v_sql;
    – Inicializar centroides iniciales (tomamos por ejemplo los dos primeros)
        FOR i IN 1..k LOOP
            FETCH v_data_cursor INTO v_centroid(1),v_centroid(2);
            INSERT into CENTROIDS VALUES (v_centroid(1),v_centroid(2),i);
        END LOOP;
        CLOSE v_data_cursor;
    – Bucle principal del algoritmo
        v_sql := 'SELECT id, '||column1||', '||column2||' FROM ' || tabla;
        OPEN v_data_cursor FOR v_sql;
        WHILE v_changed LOOP
            v_changed := FALSE;
    – Asignar puntos a los clústeres
            LOOP
                FETCH v_data_cursor INTO v_fila(1),v_fila(2),v_fila(3);
                EXIT WHEN v_data_cursor
                SELECT CLUSTER_COLUMN INTO v_cluster FROM CLUSTERS WHERE id = v_fila(1);
                v_punto:=centroidArrayType(v_fila(2), v_fila(3));
                IF assign_cluster(v_punto) != v_cluster THEN
                    v_changed := TRUE;
                    v_cluster_nuevo:=assign_cluster(v_punto);
                    UPDATE CLUSTERS
                    SET cluster_column = v_cluster_nuevo
```

```
                WHERE clusters.id = v_fila(1);
            END IF;
        END LOOP;
    – Actualizar centroides
        FOR j IN 1..k LOOP
            v_sql2:='SELECT AVG('||column1||'), AVG('||column2||') FROM '||tabla||'WHERE id IN
(SELECT id FROM CLUSTERS WHERE cluster_column = '||j||')';
            OPEN v_data_cursor2 FOR v_sql2;
            FETCH v_data_cursor2 INTO v_centroid2(1),v_centroid2(2);
            UPDATE CENTROIDS
            SET COLUMN1 = v_centroid2(1)
            WHERE cluster_asignado=j;
            UPDATE CENTROIDS
            SET COLUMN2 = v_centroid2(2)
            WHERE cluster_asignado=j;
            CLOSE v_data_cursor2;
        END LOOP;
    END LOOP;
    CLOSE v_data_cursor;
    DBMS_OUTPUT.PUT_LINE('ASIGNACIÓN DE CLUSTERS');
    FOR n in 1..k LOOP
        DBMS_OUTPUT.PUT_LINE('————————————————————');
        DBMS_OUTPUT.PUT_LINE('Los elementos del cluster '||n||' son:');
        FOR tupla in (SELECT * FROM CLUSTERS) LOOP
            IF (tupla.cluster_column=n) THEN
                DBMS_OUTPUT.PUT_LINE(' ID '||tupla.id);
            END IF;
        END LOOP;
    END LOOP;
    DELETE FROM CLUSTERS;
    DELETE FROM CENTROIDS;
    COMMIT;
END k_means;
```

A.4.1 Subalgoritmos utilizados en Algoritmo K-Medias

A.4.1.1 euclidean_distance

```
create or replace FUNCTION euclidean_distance(p_point1 centroidArrayType, p_point2 centroidA-
rrayType) RETURN NUMBER
IS
    v_distance NUMBER := 0;
BEGIN
    FOR i IN 1..p_point1.COUNT LOOP
        v_distance := v_distance + POWER(p_point1(i) - p_point2(i), 2);
    END LOOP;
    RETURN SQRT(v_distance);
END euclidean_distance;
```

A.4.1.2 assign_cluster

```
create or replace FUNCTION assign_cluster(p_punto centroidArrayType) RETURN NUMBER
IS
    p_centroid centroidArrayType:=centroidArrayType(null, null);
    v_min NUMBER;
    v_cluster NUMBER := 1;
BEGIN
    EXECUTE IMMEDIATE 'SELECT COLUMN1, COLUMN2 FROM CENTROIDS WHERE
CLUSTER_ASIGNADO = 1'
    INTO p_centroid(1),p_centroid(2);
    v_min := euclidean_distance(p_punto, p_centroid);
    FOR centroide in (SELECT COLUMN1, COLUMN2, CLUSTER_ASIGNADO FROM CEN-
TROIDS) LOOP
        p_centroid:=centroidArrayType(centroide.COLUMN1, centroide.COLUMN2);
        IF euclidean_distance(p_punto, p_centroid) <v_min THEN
            v_min := euclidean_distance(p_punto, p_centroid);
            v_cluster := centroide.CLUSTER_ASIGNADO;
        END IF;
    END LOOP;
    RETURN v_cluster;
END assign_cluster;
```

A.4.2 Tablas auxiliares para Algoritmo K-medias

```
create TABLE CENTROIDS (COLUMN1 in NUMBER, COLUMN2 in NUMBER, CLUSTER_ASIGNADO
in NUMBER);
create TABLE CLUSTERS (ID in NUMBER, CLUSTER_COLUMN in NUMBER);
```